ASMO: Autonomous System for Mowing Operations

Christopher Allen

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2018

Copyright © 2018 Christopher Allen

Abstract

The goal of this project is to create a heavy-duty autonomous robotic system for mowing complex lawns. To this end, I will mechanize a commercial-grade 27 HP zero-turn mower and automate its control with a computer. The mower will be fully outfitted with various sensors and safety protocols to allow it to mow without operator intervention. The mower will include a custom-built laser guidance system as its primary way of navigating the environment. Software will be written to virtualize multiple laser units into one unit to provide a 360-degree ground level view of the environment. The mower will also include a novel custom-built wheel odometry system impervious to dirt and dust. Software will be written to fuse data from multiple sensor systems to provide accurate odometry data for use in navigation. The navigation system will incorporate a unique custom virtualized mowing environment for path planning purposes.

Current homeowner robotic mowers are small and require the installation of wires to guide the robot's path. Current commercial robotic mowers generally require a remote operator to control the unit. In contrast, ASMO will be a large, commercial grade mower, operating autonomously, without guide wires, inside the work area. This project will develop both software and hardware systems to meet this goal.

Frontispiece

Acknowledgments

I want to thank my thesis advisor, Dr. James L. Frankel. His insight and advice throughout this project has been invaluable. Dr. Frankel's course, CSCI E-92 Principles of Operating Systems, has been a continued inspiration for me both personally and professionally. His expertise, love of the subject matter, and dedication to his students is truly impressive. There are not many educators that would work late into the night with their students to ensure the success of their projects.

I would also like to thank my father. It was his influence early on that inspired my interest in all things computer related. His advice and encouragement throughout this project has been a great help.

Lastly, I would like to thank my wife. Her love and support throughout our life together has been unwavering. Her willingness to endure my many late nights and weekends working on this project is most appreciated.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mowing a residential yard is a time-consuming process. It can also be problematic for those of us allergic to insect bites, particularly wasps and ground-based hornets. Several years ago, I had to make three separate trips to the emergency room one summer due to yellow jacket stings. I accidentally mowed over their nests, which they did not appreciate. Since then, I have used a landscaper to mow the lawn. However, I've always thought that, as a programmer pursing a Software Engineering Master's degree and an avid robotics enthusiast, there could be more cost-effective and interesting alternatives. Thus, the Automated System for Mowing Operations, or more simply ASMO, was born.

The goal of this project was to explore robotics and artificial intelligence systems by creating a cost-effective system capable of mowing my 1.33-acre yard without operator intervention. To this end, I purchased a 27 HP zero-turn mower with the intention of first mechanizing it and then programming it, to autonomously mow my yard. Zero-turn mowers utilize a differential drive that allows them to pivot on center. The work area in question is strewn with pitfalls such as stone walls, gardens, trees, and steep inclines, so the zero-turn capability allows tight maneuvering around such obstacles. In creating a system capable of mowing this area, many interesting aspects of computer science will be explored. In the end, perhaps some useful algorithms and code can be contributed back to society in the form of open source software.

ASMO is based on the 2016 commercial-grade, 27 HP Outlaw XP Mower by Bad Boy Mowers of Arkansas, USA. I mechanized this mower and installed an on-board computer system for control purposes. The software required to run the system was implemented using the Linux-based Robot Operating System (ROS). ROS is an open-source framework written in C with a wide variety of university, commercial, and governmental contributors. Using ROS allowed me to leverage the work and research done by these contributors. For instance, the ROS core provided a pre-built messaging system that I leveraged for inter-process communication between services. ROS allowed me more time to focus on my specific task, which was to do the research and development necessary to create a system capable of mowing a complex yard in an efficient manner, utilizing a custom-built laser guidance system as a primary means of navigation.

The mower has been outfitted with various sensors to provide information about the operating environment. Sensors consist of GPS, laser-based distance sensors, accelerometers, gyroscopes, magnetometers, temperature sensors, and hall-effect sensors. Software has been written that plugs into ROS, providing analysis of sensor information for path planning purposes. I implemented a Boustrophedon path planning algorithm in ASMO for the efficient mowing of lawns. I utilized techniques of Cellular Decomposition to plan paths around localized obstacles. The path planning algorithm I employed in ASMO allows for efficient mowing of the lawn, which means mowing in straight lines, handling itself well on inclines, and avoiding both dynamic obstacles such as small animals and people as well as static obstacles such as rocks and other unexpected though non-moving objects. Most of the robotic mowers today are small and

require a guide wire to be installed in the ground. Additionally, they tend to mow in random patterns. ASMO is a commercial grade mower capable of handling much larger areas without the need for modifying its operating environment.

This section describes the project background, the hardware and the software system requirements of the ASMO, as well as the outline of the thesis itself.

1.1 Background

There are a variety of robot mowers on the market today. They tend to exist at two ends of a spectrum. At one end are the smaller, lower cost, robot mowers meant for small scale residential mowing (up to 1.5 acres generally). At the other end are the industrial strength robot mowers meant for commercial duty. The residential mowers are fairly autonomous, generally guided by a user-installed electric wire that marks the perimeter of the mowing area. These residential robotic mowers then have a simple random mowing pattern inside the area defined by the guide-wire. This random mowing pattern is similar to the vacuuming pattern found in the popular Roomba robot vacuum (Galceran, 2013). The commercial mowers are much larger and remotely controlled by an operator. The RG3 autonomous mower by Cub Cadet is the only autonomous commercial mower available today.

There is currently a great deal of commercial research taking place regarding robotic mowing. iRobot, the makers of the popular Roomba robotic vacuum, recently received government approval for making a robot lawn mower (Wollerton, 2015). Briggs & Stratton is also exploring robotic mowing (Held, 2015). These are in addition to the various companies already producing robotic mowers (see a list below).

In addition, autonomous and robotic mowers have been sought as a possible way to help solve the labor crisis currently plaguing the landscape industry (Stewart, 2016). A leading example is the Valley Brook Country Club in Pennsylvania, which was able to find significant cost savings through the deployment of the Cub Cadet RG3 autonomous mower (Reitman, 2015). But it's not just cost savings for clients that is driving research forward, there are also safety benefits. For instance, Dixie Lawn Service in Kings Mountain, North Carolina, was able to reduce the number of workers sent to dangerous job sites—sites with steep slopes requiring the use of chainsaws and other high-powered equipment—to two with the use of a TRAXX RF Mower (Hall, 2016).

1.1.1 Residential Systems

Below is a survey of the various systems on the market today designed for small-scale mowing operations in residential environments.

1.1.1.1 Husqvarna Automower 450X

The Husqvarna Automower 450X (Figure 1) is a top-of-the-line robot mower. It is a small mower with a 9.45" cut width. It is powered by a lithium-ion battery and meant to cut areas less than 1.25 acres. It has a semi-random, GPS-assisted, mowing pattern with a typical run time of 260 minutes per charge and takes 75 minutes to recharge. This mower has an on-board GPS to assist in navigation but also uses boundary and guide wires to limit the mowing area.

Figure 1. Husqvarna Automower 450X.

Lithium battery with a 9.45" cut and a retail price of $3,499.95

1.1.1.2 Robomow RS630

This is the top-of-the-line robot mower from Robomow. It is a small mower with a 22" cut width. It is powered by a lithium battery and meant to cut areas less than 0.75 acres. It has a random mowing pattern with a typical run time of 50-70 minutes per charge and takes 120 minutes to recharge. This mower uses a perimeter wire to limit the mowing area.

Figure 2. Robomow RS630.

Lithium battery with a 22" cut and a retail price of $2,499.00

1.1.1.3 John Deere Tango E5

This is the only robotic mower sold by John Deere. It is a small mower with an 11.8" cut width. It is powered by a lithium-ion battery and meant to cut areas less than 0.5 acres. It has a random mowing pattern with a typical run time of 60 minutes and a typical recharge time of 80 minutes. This mower uses a perimeter wire to limit the mowing area.

Figure 3. John Deere Tango E5.

Lithium battery with a 12" cut and a retail price of $3,028.00

1.1.1.4 Bosch Indego 1000

This is the top-of-the-line robotic mower by Bosch. It is a small mower with a 10.24" cut width. It is powered by a lithium-ion battery and meant to cut areas less than 0.25 acres. It has a random mowing pattern with a typical run time of 50 minutes with a typical recharge time of 50 minutes. This mower uses a perimeter wire to limit the mowing area.

Figure 4. Bosch Indego 1000.

Lithium battery with a 10.24" cut and a retail price of £850.00

1.1.1.5 Denna L1000A

This is the top-of-the-line robotic mower by Chinese manufacturer Hangzhou Favor Robot Technology Company. It is a small mower with a 9.45" cut width. It is powered by a lithium battery and meant to cut areas less than 1 acre. It has a random mowing pattern with a typical run time of 180 minutes with an unspecified recharge time. This mower uses a perimeter wire to limit the mowing area.

Figure 5. Denna L1000A.

Lithium battery with a 9.45" cut and a retail price of $1,084.00

1.1.1.6 LawnBott LB300EL

This is the top-of-the-line robotic mower sold by LawnBott. It is a small mower with a 14" cut width. It is powered by a lithium-ion battery and meant to cut areas less than 2 acres. It has a random mowing pattern with a typical run time of 5 hours with a typical recharge time of 5 hours. This mower uses a perimeter wire to limit the mowing area.

Figure 6. LawnBott LB300EL.

Lithium battery with a 14.00" cut and a retail price of $4,999.00

1.1.1.7 Worx Landroid

This is the only robotic mower sold by Worx. It is a small mower with a 7" cut width. It is powered by a lithium-ion battery and meant to cut areas less than 2.6 acres. It has a random mowing pattern with an unspecified run time and a typical recharge time of 1.5 hours. 2.6 acres is substantially higher than other similar mowers and seems to be largely marketing hype given the small 7" cut-width. As the typical run-time could not be located, given other robot mowers, it may be assumed that the typical run-time is equal to or less than the recharge time, meaning the average run time may be around 1.5 hours. Page 11 of the operator's manual states that it will take 27 actual working hours to mow

the maximum area of 2.6 acres. This mower uses a perimeter wire to limit the mowing area.

Figure 7. Worx Landroid.

Lithium battery with a 7.00" cut and a retail price of $996.96

1.1.2 Commercial Systems

Below is a survey of the various commercial-grade industrial robotic mowers available on the market today. These systems are much more robust than the electric residential systems previously discussed. Most of the commercial-grade systems require remote control by a human operator. In these cases, it is expected the operator maintain line-of-sight with the mower as none of the remotely controlled systems discussed below provide support for a video feed.

1.1.2.1 EvaTech Goat

The EvaTech Goat (Figure 8) is marketed as a commercial grade mower. It has a large 32" cut and is powered by both gasoline as well as electricity. The mower deck is powered by a 6.75 HP gasoline engine while the rear wheels are driven by battery-

powered wheelchair motors. The batteries recharge themselves whenever the gasoline engine is running using a patented hybrid system. This mower is controlled remotely by a human operator.

Figure 8. EvaTech Goat.

6.75 HP Gasoline/Electric Hybrid, 32" cut and a retail price of $4,900

1.1.2.2 Superdroid Robot Lawn Mower Kit

The Superdroid Robot mower (Figure 9) is sold as a kit. The kit contains a modified 66-inch Swisher Tow-Behind Mower and is driven by two wheelchair motors. This mower is large, weighing in at 600 pounds. The mower is powered by a 500-cc gasoline engine and the drive motors are powered by four 35 Ah 12V batteries. The batteries are not charged automatically by the engine but instead must be manually charged using a separate battery charger. This mower is controlled remotely by a human operator.

Figure 9. Superdroid Robot Lawn Mower Kit.

5 HP Gasoline/Electric Hybrid with a 66" cut and a retail price of $6,590

1.1.2.3 Summit Mowers TRX-44 PRO

The TRX-44 PRO (Figure 10) is the top-of-the-line mower from Summit Mowers. Summit Mowers makes several different "slope" mowers that are specially designed for mowing on steep grades (up-to 50 degrees). These mowers are zero-turn mowers that have been modified to include the track and remote-control mechanisms. The mower is powered by a 24 HP gasoline engine and has a cutting width of 42". The mower is remotely controlled by a human operator. Summit Mowers also sells a conversion kit for turning an existing mower into a remote-control slope mower. The remote-control kit sells for $715 while the track conversion kit sells for $5,500.

Figure 10. Summit Mowers TRX-44 PRO.

24 HP Gasoline with a 42" cut and a retail price of $22,500

1.1.2.4 Deltrak 2.5 Flail Mower

The Deltrack 2.5 is the top-of-the-line industrial slope mower from Irus. It is powered by a 38 HP Turbocharged gasoline engine. It has a cut width of 55" and is driven by hydraulic motors. This mower is designed for heavy grass and brush. The mower is remotely controlled by a human operator.

Figure 11. Deltrak 2.5 Flail Mower.

38 HP Diesel with a 55" cut and a retail price of $30,000

1.1.2.5 Alamo Industrial Traxx RF Flail Mower

The Traxx RF is another heavy-duty slope mower for grades up to 60 degrees. It has a 51" cut and is powered by a 40 HP diesel engine. This mower is designed for heavy grass and brush. The mower is remotely controlled by a human operator.

Figure 12. Alamo Industrial Traxx RF Flail Mower.

40 HP Diesel with a 51" cut and a retail price of $78,665

1.1.2.6 Cub Cadet RG3

The RG3 is an autonomous mower sold by Cub Cadet and meant for mowing golf courses. This mower seems to be in the research or beta testing phase and technical information is scarce. While this mower is marketed as an autonomous mower, it appears to be far from simple and requires a significant investment in time to both install and maintain. In the article entitled, "Do robotic mowers dream of electric turf?", Paul Grayson says the following:

The price of an individual RG3 does not reflect the installed cost of the system. For an 18-hole course with a 19th green for putting, the installed cost of the entire system is about $225,000. Once installed, it requires two full-time employees to operate. While this price sounds outrageous, properly applied the system pays for itself in four years, and then continues to save the course money (Grayson).

Figure 13. Cub Cadet RG3.

Unspecified power system and unspecified cut with a retail price of $45,000

1.1.3 Homemade Systems

In addition to the commercially available residential and industrial mowers, there are also many do-it-yourself kits and instructional sites available that detail the conversion of existing push mowers into remote controlled mowers. There is generally

no attempt made to automate the mowing operations in these projects. Rather the focus is on remote control of the machines by a human operator. In this sense they are similar to the commercial systems described in the previous section. This section provides a survey of these projects.

1.1.3.1 Arduino R/C Lawnmower

This remote-controlled mower consists of a gasoline-powered mower that has been modified with the addition of battery-powered wheelchair motors. This was originally published as a set of instructions on the popular Instructables site (Warren, 2010), shortly followed by another write-up in MAKE magazine (Warren, 2012). The author, John-David Warren, followed up with a variation of this approach in his book Arduino Robotics. This approach is similar to the EvaTech Goat mower listed above.

Figure 14. Arduino R/C Lawnmower.

Remote control lawnmower created by Instructables user johndavid400

1.1.3.2 DIY R/C Lawnmower

This is another example of a push mower converted into a remote-controlled mower using off-the-shelf radio-controlled equipment and wheelchair motors.

Figure 15. DIY R/C Lawnmower.

Remote control lawnmower created by Terry Creer

1.1.3.3 DIY R/C Lawnmower

The video clip shown in Figure 16 shows a mower that is unique in that it uses gyros to help keep the mower moving in a straight line when mowing remotely.

Figure 16. DIY Semi-Autonomous R/C lawnmower.

Semi-autonomous remote-control lawnmower created by YouTube user Dale Z Jr

1.1.3.4 Remote-controlled lawnmower

The video clip shown in Figure 17 illustrates another example of the conversion of a push mower to a remote-control mower. The video has a nice explanation of the radio-controlled parts.

Figure 17. Remote-controlled lawnmower.

Push mower conversion to remote-control lawnmower by YouTube user Mr2Tuff2

1.1.3.5 Minimalize DIY Conversion

The video clip shown in Figure 18 shows a minimalist approach to modifying an existing push mower. The existing mower is kept intact with a rear drive section bolted on where the push handle usually goes. The effectiveness of the mower is reduced in that the front wheels are kept intact as opposed to having been replaced with the usual casters. This approach would cause the front wheels of the mower to scrape the ground when turning. This is not an issue with manually operated mowers as the operator would push down on the mower's handle, lifting the front wheels, then pivoting on the back wheels to perform

the turn. The scraping is not evidenced in the video as the video only shows the mower going straight. Still, it is an interesting quick and dirty approach.

Figure 18. Minimalist DIY Conversion.

Remote Control Mower created by YouTube user Lee Evans

1.1.3.6 All Terrain Track Chair

The popular approach shown in previous YouTube conversions is to take an existing gasoline-powered mower and motorize it using a pair of wheelchair motors powered by 12 or 24 VDC batteries. The approach taken with the All Terrain Track Chair (Figure 19) is closer to the approach I am taking in this project. Kustoms By Kent took a diesel-powered Bad Boy Outlaw Mower and turned it into an All-Terrain Wheelchair. It can be controlled remotely but can also have an operator sitting locally on the unit. This project is a work-in-progress.

Figure 19. All Terrain Track Chair (ATTC).

Conversion of a Bad Boy Diesel mower to remote-control by Kustoms By Kent

1.1.4 Academic Research

The general approach in the various residential systems is to use a guide wire to create a boundary area and then allow the robot to navigate freely inside the boundary area cutting the area randomly. Obstacle avoidance is generally performed using bump sensors. This approach works well for smaller robots but would not work for the more robust machines needed to cut larger areas. As such, the larger commercial units tend to be remotely controlled by an operator. Academic research is focused on more autonomous ways of performing mowing and farming operations. However, the approaches taken in academic environments tend to be very costly and as such would not work, without modification, in a low-cost residential setting. As the goal of the ASMO project is to bring autonomous mowing operations to residential areas, I investigated a variety of academic approaches to autonomous navigation to see if there might be a way to adapt them to be more cost-effective.

The Institute of Navigation (ION) held autonomous mower competitions between 2005 and 2012. Various universities participated in this competition over the years, producing a wealth of useful information and experience on autonomous mowing operations. As noted by Hughes, "the majority of the teams at the ION competition utilize RTK GPS receivers to enable precise robot navigation" (Hughes, 2010). RTK GPS offers centimeter level accuracy—much better than the 3-meter accuracy of a normal low-cost GPS receiver—but costs thousands of dollars. While RTK GPS was a general approach used for planning operations, laser-based distancing systems were used as the general approach for short-term obstacle avoidance. Such an approach is much less dangerous than bump-sensors for high-power machines. The LMS series of laser scanners manufactured by SICK were used by a number of the teams (Hughes, 2010; Woodall, 2012) in the competition to detect and avoid short-term obstacles. Unfortunately, these laser systems also cost several thousand dollars. Neither of these approaches seemed suited, out-of-the-box, to help develop a low-cost system for automating mowing operations, which is the goal of this project.

In the absence of a sensor system capable of providing absolute positioning information, I looked for a way to estimate the robot's position. State estimation is a general term used to refer to the process of combining various sensor inputs together in order to generate an approximation of the actual state. In the case of ASMO I needed a low-cost way of estimating the robot's position for path planning purposes.

The traditional technique of determining a robot's position is to use wheel encoders. This is because encoders are an easy and inexpensive way of measuring wheel rotation. Since the wheel is of a known size, the distance travelled can be computed

using the circumference of the wheel multiplied by the number of turns being made, as measured by the encoder. The problem with this approach is that errors accumulate over time due to wheel slippage. Kreinar detailed the various types of Odometry errors in chapter 2.3 of his paper entitled "Filter-Based Slip Detection for a Complete Coverage Robot" (Kreinar, 2013). Essentially, he defined wheel slip as "any encoder measurement where the measured wheel velocity does not equal the true wheel velocity" (p.43). Kreinar's solution to account for wheel slip was to combine measurements from an IMU with those of RTK GPS and wheel encoders to more accurately compute wheel odometry where wheel slip is taken into account. He successfully combined these sensors together using an augmented Extended Kalman Filter, resulting in accurate positioning information over time. I will also be taking this approach, but instead will be de-emphasizing the GPS measurements as the GPS utilized in this project does not have the accuracy of RTK. Another difference between Kreiner's approach and my own is that the wheel encoders used in his project had 24,000 ticks-per-wheel revolution. This was due to the use of an electric motor where a precise encoder could be attached. In the ASMO project, magnets are being retrofitted onto an existing wheel rotor, providing a precision of 45 ticks per revolution.

In order to plan a path, the first step is to have a map of the environment. In the ION competition, a map of the environment was generated prior to the actual competition by driving the robot around the competition area and using the SICK laser-based distancing system to take measurements of the area. As previously mentioned, the SICK LMS system is quite expensive, costing upwards of $5,000. As the ASMO project is meant to be a low-cost solution, expensive lasers were out of the question. Instead,

inexpensive discrete lasers from Garmin were combined with other inexpensive components to create a low-cost Light Detection and Ranging (LIDAR) system. This LIDAR operates at 2 Hz instead of the faster 40 Hz of the more expensive units, but the advantage is they cost $200 instead of $5,000.

Simultaneous Location and Mapping (SLAM) is a common technique of creating a map of an environment while locating the robot in the environment—in real time. It was outlined by John Leonard and Hugh Durrant-Whyte in their paper entitled "Mobile Robot Localization by Tracking Geometric Beacons" (Leonard, 1991). SLAM operates by taking distance measurements from a 2D plane and looking for landmarks in the data. The robot position is estimated based on its relation to these various landmarks. SLAM consists of several smaller steps such as landmark extraction, data matching, and state estimation. The ROS system used in this project contains implementations of two popular SLAM algorithms.

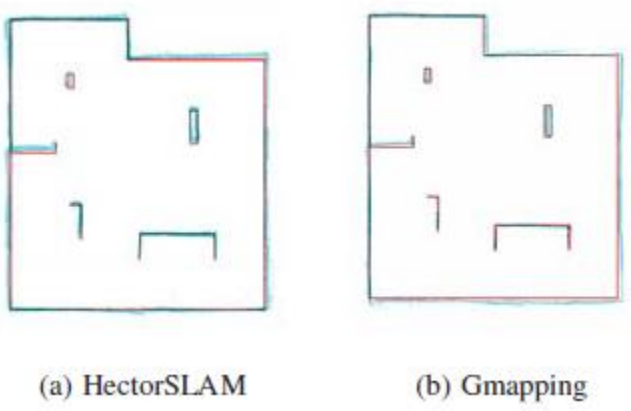
The first is Hector SLAM. Hector SLAM takes advantage of faster LIDAR systems to create a position estimates based on an inertial sensing system created by comparing the difference between scans at a very rapid rate. Since the LIDAR used on ASMO will only operate at 2 Hz, this approach will not be used as it is not fast enough for the Hector SLAM algorithm to operate effectively. Instead, the slightly less accurate Gmapping algorithm will be used. The Gmapping algorithm, as defined by Grisetti *et al.* (Grisetti, 2007), combines laser-based distance measurements with odometry data to assist in state estimation. The Hector SLAM algorithm does not make use of odometry data. The Hector algorithm, while slightly more accurate, would be much more

beneficial in an aerial environment where odometry data is not present. Table 1 presents a comparison of the errors from real world tests of these two algorithms (Santos, 2013):

Table 1. Error estimations of three separate tests of real world experiments

Test Number	HectorSLAM	Gmapping
1	1.1972	2.1716
2	0.5094	0.6945
3	1.0656	1.6354

As can be seen in Figure 20, the Gmapping algorithm was only slightly less effective than the HectorSLAM algorithm. Both algorithms operated successfully in the real-world tests performed by Santos, *et al.*



(a) HectorSLAM (b) Gmapping

Figure 20. Real World Tests of SLAM algorithms (Santos, 2013)

Red represents the ground truth and blue the final map.

Laser-based SLAM techniques have an added advantage over RTK GPS for path planning purposes. Laser-based SLAM techniques make use of distance measurements

of the environment captured in real-time. As such, they can be used to detect dynamic obstacles that weren't present when the map was made. RTK GPS can be used to avoid obstacles known when the map was made but has no way of detecting a dynamic obstacle, such as a person, entering the path of the mower.

Lasers used for purposes of SLAM and obstacle detection are generally mounted on the front of the robot. The SICK LMS lasers generally have a field of view from 100 to 180 degrees (Mertz, 2013). As such, they cannot capture an entire 360-degree view of the environment. Since the SLAM algorithms operate by identifying landmarks in the surrounding environment, a mower moving in normal straight-line patterns would be somewhat limited in the landmarks that could be identified. While the LIDAR system used in the ASMO project operates at 2 Hz, there will be three LIDAR systems created and mounted around the perimeter of ASMO. The readings from these three LIDAR systems will be combined to form a virtual LIDAR unit that should be much more accurate than a single unit. The anticipation is that a 360-degree view will provide the opportunities for many more landmarks to be identified by the SLAM algorithm, thereby making up in part for its slower speed. Additionally, since the front LIDAR units cover the same forward-facing area, the data from the sensors can be interleaved and the individual outputs from the front LIDAR units can be used to detect obstacles at a rate greater than 2 Hz.

As previously mentioned, the various autonomous lawnmowers available commercially today are penned by a buried boundary wire and tend to mow in a random pattern, much like an indoor robot vacuum cleaner. In order to achieve the more predictable, and thereby safer, aesthetically pleasing parallel cut lines made by human

operators, another approach is required. The robot described in the paper "An Obstacle-Edging Reflex for an Autonomous Lawnmower" took 1st place in the 2009 ION Autonomous Mower competition (Daltorio, 2010). The paper describes a unique, low-level, method for avoiding obstacles while maintaining a high-quality cut—specifically edging closely around obstacles. As with the ASMO project, their robot also uses a combination of LIDAR and GPS for positioning.

There are several academic projects where an existing mower is mechanized for use in a robotic system. In the paper entitled "Autonomous Robot in Agriculture Tasks" (García-Alegre, 2017) discuss the conversion of a commercially available riding lawn-mower into an automated mowing and spraying robot. This project is similar to the ASMO project but operates on flat areas as opposed to inclines. That said, the authors do discuss unique ways of localizing and precision positioning of the robot system during operation that may be applicable to the ASMO project. In the paper entitled "A Tele-Autonomous Heavy Duty Robotic Lawn Mower" (Jarvis, 2001), Jarvis discusses a unique tri-level control strategy for a heavy-duty mowing machine. It makes use of many of the same sensor types used in the ASMO project. The tri-level system is a combination of a lower-level behavior-based reactionary control system and a higher-level planning system. Behavior-based reactionary control means that the robot will perform a prescribed behavior in reaction to some external stimuli. This approach is similar to the approach planned for the ASMO control system, specifically as it relates to controlling the system behavior for short-term path planning.

Long-term path planning involves making a qualitative or quantitative determination as to how to navigate from one location to another so that no collisions

occur along the way. Long-term path planning is typically performed by a global-planner, which is designed to take the entire workspace into account when developing a plan to meet a particular goal. This is different from short-term path planning as long-term path planning involves the over-arching goal, not just an immediate behavior-based response to a dynamically arising situation. Short-term path planning is typically performed by a local-planner, because only the local environment is considered when formulating a plan of action.

A typical global planner approach to finding a path from point A to point B is to use a technique known as cellular decomposition. In cellular decomposition, the idea is to take a bounded area and subdivide it into smaller regions based on trapezoids or triangles. Next, the free space needs to be decomposed. This is done by placing parallel lines from each vertex of every internal polygon to the edge of the exterior bounded area. Each resulting cell is then numbered and represented as a node in a connectivity graph. The various paths from point A to point B can then be followed along the graph. This process is shown in Figure 21, as demonstrated by Latombe (Latombe, 1993). The best path can be determined based on quantitative measures such as distance or time.

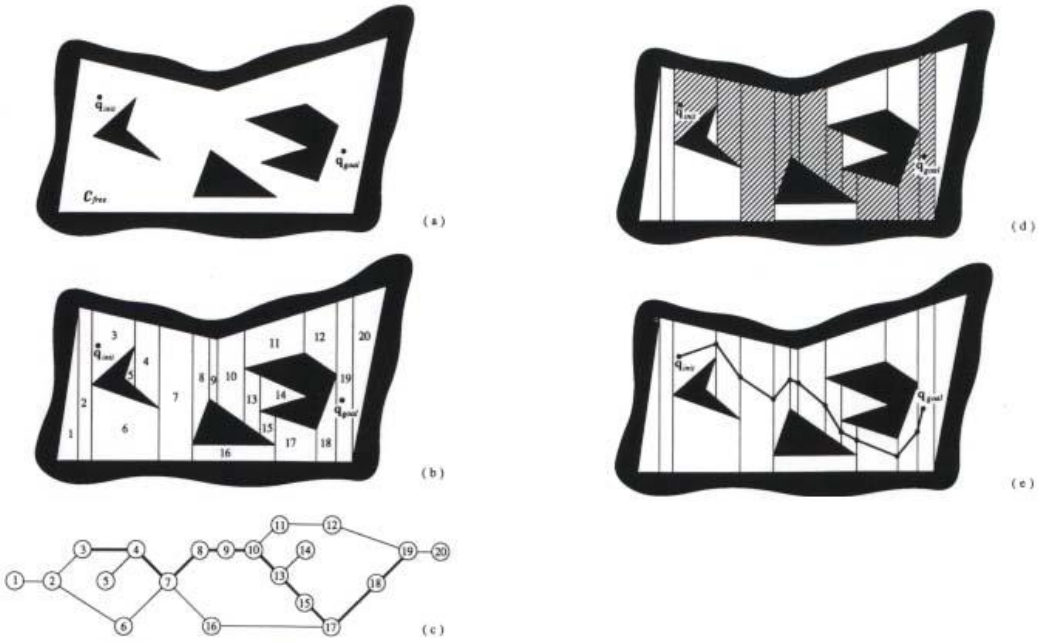


Figure 21. Cellular decomposition and the creation of a connectivity graph (Latombe, 1993)

Mowing operations require a different kind of long-term planning technique than the typical global planning algorithms required for navigating from point A to point B. In a mowing operation the goal is to cover an area completely, while avoiding collisions along the way. In this regard, mowing operations are similar to agricultural operations, where the goal is to cover an entire field for planting or harvesting purposes. In the paper entitled "Path Planning Algorithms for Agricultural Machines" (Oksanen, 2007), the authors are dedicated to finding optimal navigation paths for autonomous agricultural machines. The paper investigates two specific algorithms. The first involves dividing the work-area into trapezoidal sub-plots that are easier to manage than the whole. The second algorithm involves simulating all possible routes on the basis of the machine's current state. I hope to either apply knowledge from this paper either directly or

indirectly to the path planning node that needs to be developed for the ROS navigation stack.

Long-term path planning for coverage of an area can be performed using a specific type of cellular decomposition known as Boustrophedon. In their paper entitled "Coverage Path Planning: The Boustrophedon Cellular Decomposition" (Choset, 1998), Choset and Pignon described the details of their algorithm. The Boustrophedon Cellular Decomposition is similar to the Trapezoidal Cellular Decomposition in that both approaches decompose the free space into trapezoidal cells. However, the Boustrophedon algorithm essentially merges adjacent cells. Figure 22 shows the Trapezoidal Decomposition of a free space while Figure 23 shows this same free space decomposed using the Boustrophedon Decomposition.

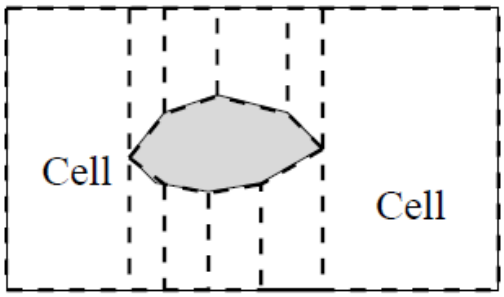


Figure 22. Trapezoidal Decomposition (Choset, 1998)

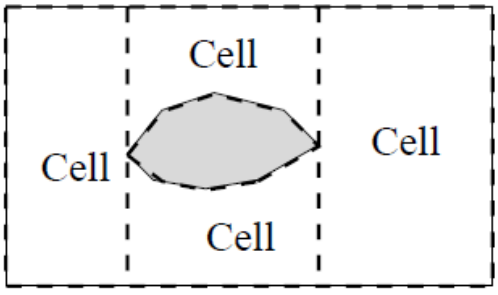


Figure 23. Boustrophedon Decomposition (Choset, 1998)

By aggregating the cells, the Boustrophedon approach reduces the number of cells that must be travelled to obtain complete coverage of an area, which increases the efficiency of the robot. The cells are aggregated by looking at the vertices on the polygons where vertical lines can be extended both up and down to the outer edges of the free space. These vertices are called critical points. Once all the critical points have been discovered, the planner computes an exhaustive walk through the adjacency graph, using a depth-first search algorithm. This process can be seen in Figure 24.

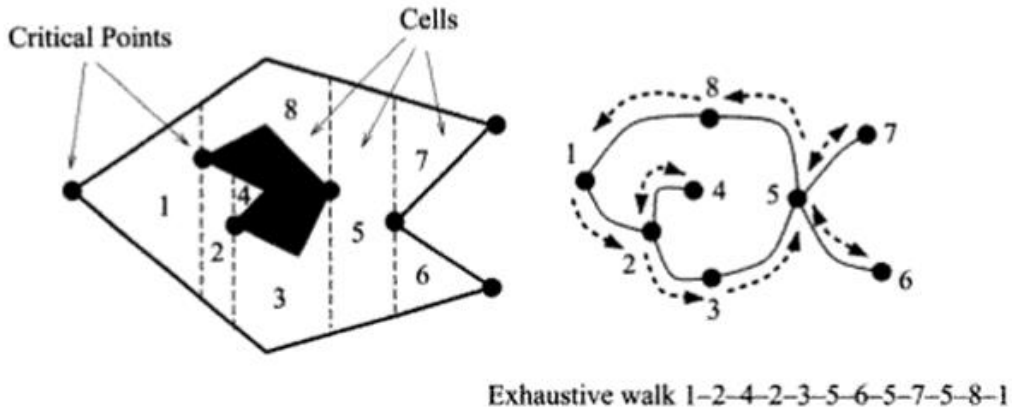


Figure 24. The Boustrophedon Decomposition of a space and its adjacency graph (Choset, 1998)

Once the exhaustive walk has been determined, complete coverage can be obtained by simply visiting each cell and performing back and forth motion (Figure 25). This motion is the "way of the ox" from which the Boustrophedon Decomposition takes its name.

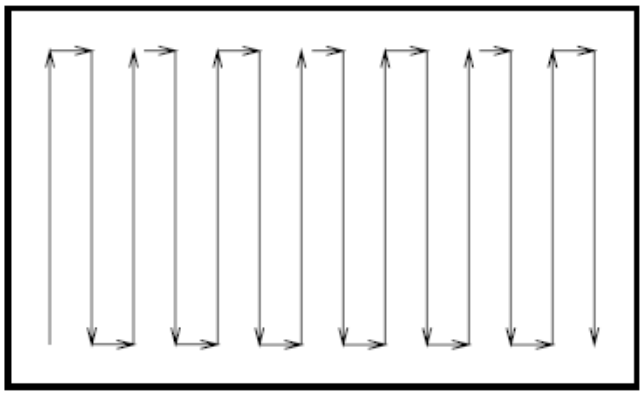


Figure 25. Boustrophedon Path (Choset, 1998)

1.2 Requirements

The main goal of this project was to develop a robust robotic system capable of autonomously mowing a large complex yard in a reasonable amount of time. One sub-goal, which is a unique aspect of this project, is to autonomously navigate primarily using a low-cost, custom built, laser guidance system. Determining the location of the robot will be supplemented with sensor data fused from hall-effect sensors mounted on the wheel rotors along with data from an Inertial Management Unit. A point cloud is created from the planar laser data and used to generate an occupancy grid of the scene in front of the robot so that an efficient path can be planned. In most autonomous systems, the point cloud needed for navigation comes from very expensive scanners capable of rendering the plane of a scene in great detail. Laser scanners capable of operating outdoors, such as the Hokuyo UTM-30LX-EQ scanner, can cost in excess of $5,000 (Acroname). By contrast, an individual LIDAR unit outlined in this thesis can be built for under $200.

As part of this process, I learned more in-depth details regarding the Linux operating system, the Robot Operating System (ROS), and programming distributed applications in C. As part of implementing ROS, custom short-term and long-term path

planners were created to handle the unique nature of efficient mowing operations, such as precise striping and mowing around trees, etc. In order to achieve these goals, there were several requirements both in hardware and software.

1.2.1 Mechanization of the Mower

The first step that I performed was to mechanize the mower itself. Larger machines capable of mowing an acre or more of grass generally require the use of gas or diesel fueled engines. These types of mowers are meant for human operation and are not designed to accommodate computer control. The act of bringing the machine under computer control is what I refer to as mechanization. There were several different components that I needed to mechanize in order for the computer to control the mower.

First, the mower's speed and direction had to be controlled by the computer. Doing so allowed the mower to follow the path determined by the computer. In order to control the amount of power available to the mower for drive and cutting operations, the throttle was also brought under computer control. Since the mower should not always be cutting grass whenever it is moving, such as when it is navigating between mowing areas, the computer also required control of the mower deck. For safety, the computer had to be able to apply the parking brake automatically when stopped as well as be able to kill the engine quickly when necessary.

1.2.2 Available Hardware

Not everything for this project had to be built from scratch. There were many pre-existing components that could be interconnected in order to form a whole operating unit. For instance, the main computer itself was purchased as a single-board computer,

which alleviated the need to purchase individual memory, video cards, and other computer sub-systems. The various microcontrollers utilized for low-level system control were also purchased as single operating units and not built from discrete chip components. The relay boards used to control on/off operations of the mower were also purchased as pre-built components that could be connected directly to the microcontrollers. Sensor boards for GPS and the Inertial Management Unit (IMU) were purchased as ready-to-use components, which meant they could be connected directly to the microcontrollers without additional hardware.

1.2.3 New Hardware Required

While many of the components required for this project were purchased as pre-built components, some of them required more customization than could be found with off-the-shelf components. For example, in order for the computer to know how fast, in what direction, and how often the drive wheels were turning, some form of encoder was required. Typical encoders are connected directly to an electric motor's shaft so when the shaft turns, light interrupts a slotted disk spinning with the motor's shaft, eventually outputting a series of 1's and 0's. Such encoders are known as optical encoders, due to their use of light. This approach wouldn't work for the mower project for a number of reasons. First, the mower uses a hydraulic motor, which has no secondary output shaft on which to connect an encoder. Attaching an optical encoder elsewhere would not be practical as it would be prone to dirt and dust build-up interfering with the breaks in the disk. Another form of encoder can utilize magnets to eventually generate the series of 1's and 0's needed to encode wheel movement. As a magnet passes in front of a sensor, the sensor detects the magnetic field and outputs a 1, as the magnet moves past the sensor,

the sensor outputs a 0. In order to utilize such a system, custom hardware needed to be created to mount the sensors and the magnets.

As previously mentioned, most LIDAR units capable of operating in harsh outdoor environments are cost-prohibitive for low-cost systems. Therefore, the LIDAR units also had to be custom built. While the laser sensor itself could be purchased as a discrete part, other parts had to be designed, purchased, and assembled to form an operating LIDAR unit. Additionally, many of the various housing and mounting components needed for this project had to be custom built from aluminum stock using a small benchtop milling machine and lathe. While 3D printing would have been easier and quicker, plastic would not have stood up to the harsh operating environment of a mower. In a mowing environment, dirt, sticks, and other material such as a stray rock, could damage sensor systems, rendering them inoperational, or worse, unpredictable.

1.2.4 Available Software

The Robot Operating System (ROS) was used as a meta-operating system sitting on-top of the Linux operating system. ROS provides many of the common functions necessary to implement robot behavior. While it provides a great number of implemented algorithms and support for pre-existing sensors, the main use of ROS in the ASMO project was for its infrastructure. By this I mean ROS has a great component-based interface where loosely coupled software components can intercommunicate. ROS also provides robust visualization and debugging packages that made the development and testing of code much simpler.

1.2.5 New Software Required

In order to support the new hardware sensor units that were built, new software components had to be created to interface to them. New ROS components were written to support the custom LIDAR units as well as the custom wheel encoder units. Additionally, new ROS components were written to interact with the various mechanized mower components such as the engine, the mower deck, the drive system, etc. New ROS components were also written to perform sensor fusion of the wheel encoders, IMU, and GPS data.

ROS excels at providing a navigational system suited for typical autonomous robot navigation where the robot needs to safely navigate from point A to point B while avoiding obstacles. However, this type of navigation does not work well for a robot that would fully mow a surface, i.e., provide complete coverage, which was needed for mowing operations. Therefore, a new global planner was implemented to support complete coverage of the working area.

1.3 Thesis Outline

Chapter 1 covered the motivation, background, and requirements of the ASMO project. This included details regarding the currently available systems for residential and commercial environments. Also covered were the various academic projects that directly relate to the current work.

Chapter 2 will dive into an overview of the system. This will include a description of the mower, its power system, and how the mower was mechanized for computer control. The very important safety system and protocols will also be covered in

this chapter. An overall description of the low-level interface to the hardware as well as the high-level software control systems will also be covered in Chapter 2.

Chapter 3 will cover the throttle control system that was developed for the ASMO project. This includes going over the details of how the throttle plate was mechanized, how the engine speed is known and how these hardware components tie into the software. Chapter 3 will also cover how the mechanical governor system was replaced with a software governor, including issues encountered during the implementation of the governing algorithm.

Chapter 4 contains a detailed explanation of the hardware and software required to create the LIDAR units. This includes coverage of the mechanical design and build process as well as how the data from the laser units was encoded for efficiency and communicated to the ROS environment. Performance of the LIDAR units is also discussed along with visual representations of the data.

Chapter 5 dives into sensor fusion and the odometer system. This includes coverage of the GPS, the IMU, and the Wheel Encoders. It also includes a discussion of the Extended Kalman Filter used to filter and fuse the data. The custom-built ROS Fused Odometry service is discussed.

Chapter 6 details the hardware and software created to allow interaction with a human operator. This includes details of the user control sub-system. Training of the mower in new environments is also covered in this chapter. Additionally, this chapter covers the process of map building.

Chapter 7 covers robot localization, which is the process of locating the robot inside a map, along with both long and short-term path planning. The algorithms

implemented for the complete coverage navigational system are detailed. There is a discussion of the software components that were written, as well as the problems encountered. The process of localization using the built-in ROS adaptive Monte Carlo Localization (amcl) service are discussed. Trapezoidal and Boustrophedon path planning algorithms are covered for the global planner. Also discussed are the various short-term path planning options tested, including how obstacle detection works and the use of ROS waypoint navigation.

Chapter 8 wraps up the thesis with a discussion of the outcome and contributions of the current work. A discussion of possible future directions and work is also outlined.

Chapter 2

System Overview

ASMO is based on the 2016 commercial-grade, 27 HP Outlaw XP Mower by Bad Boy Mowers of Arkansas, USA. This mower was mechanized and an on-board computer system installed for control purposes. The software required to run the system was implemented using the Linux-based Robot Operating System (ROS). ROS is an open-source framework written in C with a wide variety of university, commercial, and governmental contributors. Using ROS allowed me to leverage the work and research done by these contributors. For instance, the ROS core provided a pre-built messaging system that I leveraged for inter-process communication between services. ROS allowed me more time to focus on my specific task, which was to perform the research and development necessary to create a system capable of mowing a complex yard in an efficient manner, utilizing a custom-built laser guidance system as a primary means of navigation.

2.1 Description of the Hardware

ASMO has been built upon a commercial-grade zero-turn mower. I have added an on-board computer running the Linux Operating System. I have also added several on-board microcontrollers for controlling the mower operation, such as drive, steering, and the Power Take-Off (PTO) that powers the mower deck. Additionally, I have outfitted ASMO with a variety of sensors for monitoring and navigating the environment. This section describes the hardware systems of ASMO in more detail.

2.1.1 Zero-Turn Mower

The mower is arguably the most important piece of hardware for this project. Therefore, choosing the right mower was very important. There are various approaches that have been taken when creating robotic mowers to-date. The residential robotic mowers available today are not large enough to handle the planned work area. Nor are they powerful enough to handle the inclines. They are generally meant to mow small, fairly flat areas with minimal obstructions.

I did consider converting an existing push mower into a robotic mower. However, the small cutting width (generally less than 21 inches) meant the process of mowing the yard was going to take longer. There also is no suspension available on these mowers, which means more vibration will be transmitted to the sensors. It didn't take long to realize I needed a riding mower. There are two types of riding mowers available: the traditional riding mower, which uses a steering wheel (like that of a car) and a zero-turn mower, which uses two levers to provide differential steering (like that of a tank).

The zero-turn mower was an obvious choice as the differential drive system allows the mower to turn in place, simplifying the path planning process. Providing forward power to one wheel while providing reverse power to the other wheel allows the zero-turn mower to spin in place. The system can also be driven in arcs by providing more power to one wheel and less to the other. The differential drive system allowed ASMO to make faster and more accurate course adjustments. After cutting a straight line, a traditional mower has to make an arc, then find where it left off to begin cutting the other direction. A zero-turn mower can simply pivot 180-degrees around one wheel

to face the opposite direction. This greatly simplified the programming process. Additionally, mechanizing a zero-turn mower is much easier than a steering-wheel based mower as the zero-turn mowers are generally controlled using two levers to control the hydraulic fluid flowing to hydrostatic wheel drives. In the ASMO project, these levers are controlled using linear actuators, as described below.

I reviewed several different zero-turn mowers, including models from John Deere, Kubota, Hustler, and Bad Boy Mowers. After much research and deliberation, I settled on the Bad Boy Outlaw XP mower. In addition to the heavy-duty build of the mower, it also offered independent hydraulic pumps, and the easiest access to internal components. Its mower deck was also already motorized using a linear actuator for raising and lowering, allowing easy control over the height of the cut. Having independent hydraulic pumps—as opposed to having the pumps integrated into the hydraulic motors—meant they were easier to access, and therefore modify, for computer control.

Figure 26. Bad Boy Outlaw XP Mower

As can be seen in Figure 26, the Bad Boy Outlaw XP mower utilizes a fly-out design that allows all its components to be accessed very easily. The above picture demonstrates how the mower can be opened, allowing access to all its major components—including the push rods for the hydraulic pumps used in steering as well as all the relays for controlling its electrical system. Additionally, the seat swings up allowing easy access to the hydraulic pump control arms. There was room in the side arms for mounting the on-board computer. This allowed me to keep the seat in place for a human to perform debugging, manual control, and training operations.

2.1.2 Power

The Bad Boy Outlaw XP is powered by 852cc Air-Cooled Kawasaki FX engine producing 27 HP. The fuel capacity is 13.8 gallons. The mower has a 12 VDC battery and 20-amp charger built into the system. This was sufficient for powering the on-board computer system. The on-board computer requires a well-regulated power supply capable of generating clean 5 VDC. Additionally, the linear actuators and stepper motors used in this project require a well-regulated 12 VDC power supply. Two generic 12 VDC 12-amp regulators were used for this project (Figure 28) along with two generic 5 VDC 3-amp regulators (Figure 27), all ordered from Amazon.com. Both power supplies take 12 VDC direct from the mower's battery.

Figure 27. 5 VDC Regulator

Figure 28. 12 VDC Regulator

The power output was sufficient for powering both the main-board computer as well as the sensors, stepper motors, servo motors, and linear actuators. Heat dissipation and monitoring was a consideration in how the case for these power supplies was constructed. When I fabricated the case, I included two 40mm fans as well as a temperature sensor for monitoring the internal case temperature. The fans used in the cooling system are controlled by a microcontroller inside the main case, which monitors the internal case temperature. I programmed the microcontroller to control the speed of the fans based on the internal case temperature. As the temperature inside the case rises, the fan speed increases. As the temperature inside the case decreases, the fan speed decreases as well. The temperature threshold values are configurable. Additionally, if the temperature of the case rises above the configured maximum temperature, the mower system is shutdown. This process is designed to allow a controlled shutdown of the mower system rather than allow an unexpected failure to occur due to over temperature conditions.

2.1.3 Mower Control

There are various systems on the mower that had to be controlled. The following table illustrates the mower function and type of device used for its control.

Table 2. Mower Control Functions

Function	**Description**	**Device**	**Control**
Drive System Control Arms	Used to control both steering and speed. Also used for braking.	Dual linear actuators	PWM
Parking brake	Used once the mower is stopped in order to hold position while the engine is not running.	Linear actuator	Relays (x2)
Throttle	Controls the amount of power the engine generates (and thereby the maximum speed of both the PTO and drive system)	Servo	PWM
Engine start/stop	Used to start and stop the engine.	Digital Output	Relay
PTO	Used to provide power from the engine to the mower deck.	Digital Output	Relay
Safety cutoff	Used to immediate stop the engine	Digital Output	Relay

2.1.3.1 Drive System (Linear Actuators)

One of the most important functions of an autonomous mower is the ability to move the machine around the yard. A human operator controls the movement of a zero-turn mower by pushing the left and right drive arms forward or pulling them backward

(Figure 29). The control arms are connected to the hydraulic drive motors through the linkages shown in Figure 30. The left control arm is connected to left wheel's hydraulic motor and the right control arm is connected to the right wheel's hydraulic motor. Pushing a control arm forward causes the associated wheel to move clockwise, thereby driving that side of the mower forward. The more forward the control arm is moved, the faster the wheel turns. Pulling the control arm back towards the operator causes the wheel to slow from its clockwise spin until it rests at a center point where the wheel stops turning. If the control arms continue to be pulled toward the operator from the center point, the wheel begins to spin in a counter-clockwise direction, thereby causing that side of the mower to reverse direction and go backwards.

Figure 29. Drive Control Arms

Moving both control arms forward in unison causes both drive wheels to turn in the same direction at the same speed, thereby moving the mower forward in a straight line. Slowing one wheel causes the mower to turn in the direction of the slower wheel as the faster wheel drives the mower in that direction. This method of steering is known as tank drive and is common on zero-turn mowers. The zero-turn mower gets its name from the ability of spinning one wheel forward and the other wheel backwards, by pushing one control arm forward while pulling back on the other, thereby causing the mower to spin in place.

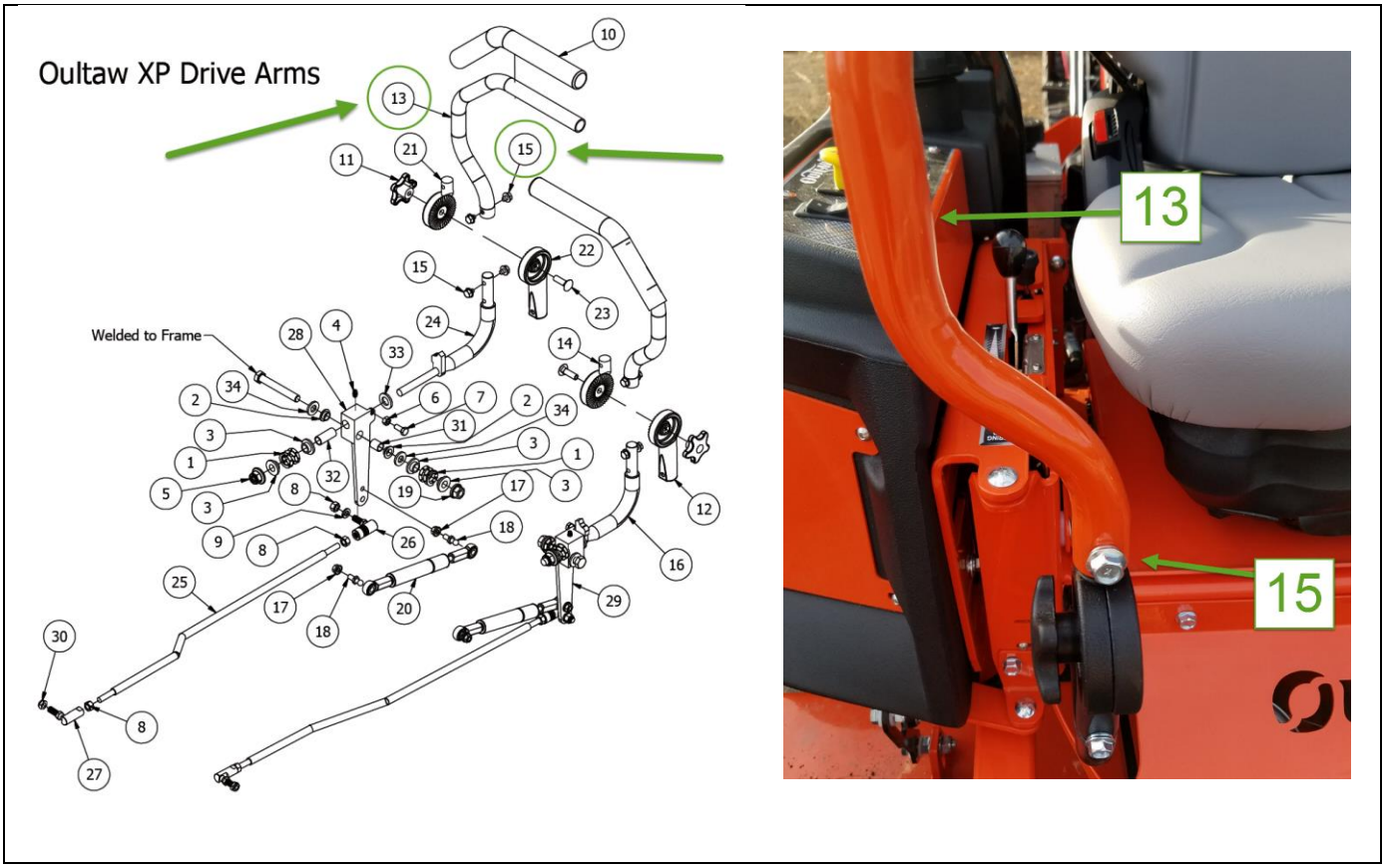


Figure 30. Drive Arm Attachment

In order for a computer to control the motion of this mower, it was necessary for it to control the movement of the control arms. Several different options were explored to make this happen. One of the reasons I chose the Bad Boy mower was because the

control arms could easily be removed. One option I considered was removing the upper part of the control arm. This could be done simply by removing a bolt designated as 15 in Figure 30. Doing so allowed the upper control arm, designated as 13 in the same figure, to be removed. Once removed, I could attach the actuator in its place. The downside to this approach was that the human operator could no longer control the mower without re-attaching the upper control arms.

I did explore the possibility of adding a joystick to allow a human operator to control the mower using the above approach. The joystick did allow easy operation in an open field. Push the joystick forward and both wheels go forward. Push the joystick up and to the left and the right wheel spins faster than the left wheel causing the mower to veer to the left. However, while the joystick did allow easy operation in an open field, it did not provide the fine-grained control necessary to maneuver the mower in tight spaces—such as back into a cluttered garage. As such, I explored other options.

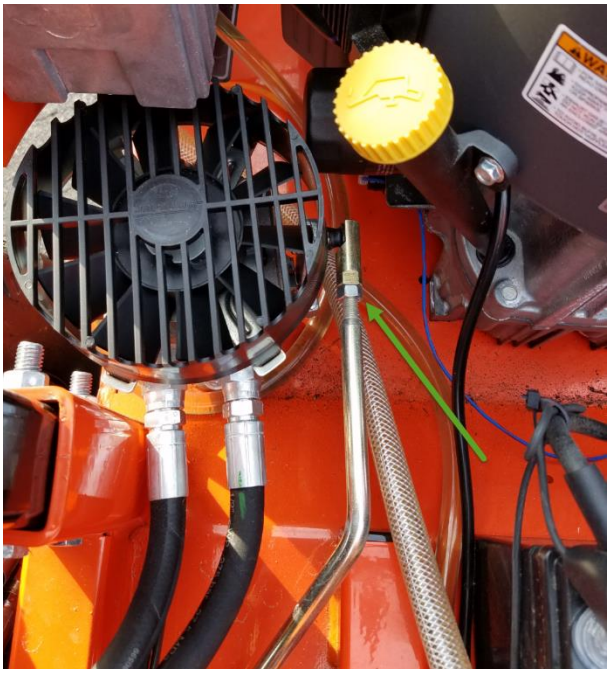


Figure 31. The green arrow points to the control arm connector

The option I settled on allowed the control arms to remain in place. Instead of removing the control arms, they were disconnected from the linkages which connected them to the hydraulic motors. I was able to purchase a replacement connector component marked 26 in Figure 30. This allowed me to connect linear actuators directly to the control arm linkages marked 25 in Figure 30. This control arm linkage was connected directly to the hydraulic motor as shown in Figure 31. However, taking this approach did require additional work in order to mount the linear actuators. I had to create special brackets to mount the actuators directly to the frame of the mower (Figure 32). Then I had to create a support bar to prevent the end of the actuator from twisting about its center as it moved in and out. The completed bracket and support bar can be seen in Figure 33.

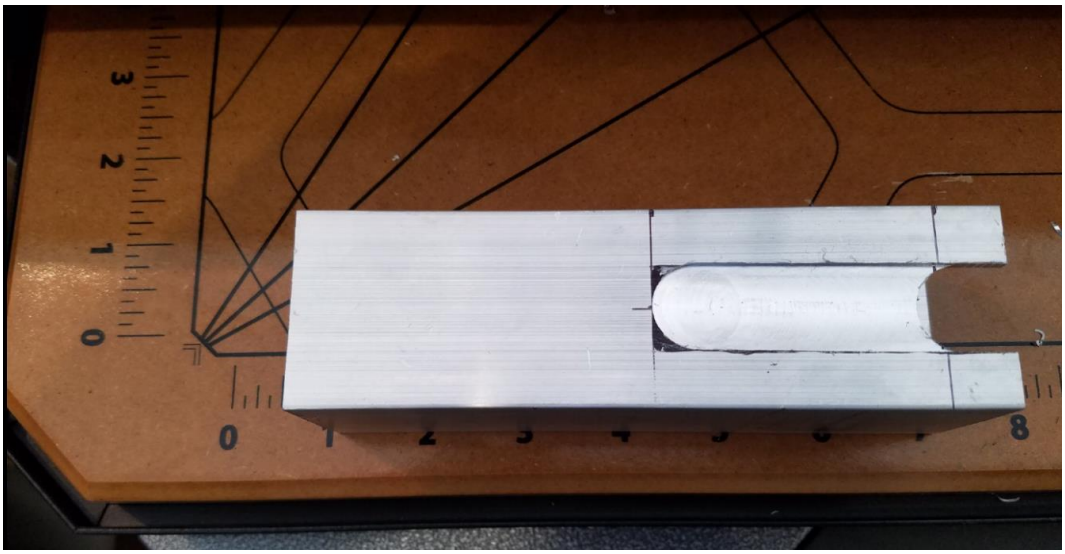


Figure 32. Milled actuator bracket

Figure 33. Completed drive actuator

During the build process, I tested the operation of the actuator on my lab bench (Figure 34). This allowed me to work through the control process, ensuring the smooth operation of the actuator. This is how I discovered the need for the support bar as the end of the actuator wanted to turn as the rod moved in and out. Normally this isn't an issue as the end of the actuator is attached to an immovable part. However, the control arm the actuator is connected to is not fixed in place but has some amount of freedom. The twisting of the actuator caused the control arm to get caught on wires. This is why the support bar was added. I fabricated the support bar out of nylon, for its durability and low friction, and corrosion resistance.

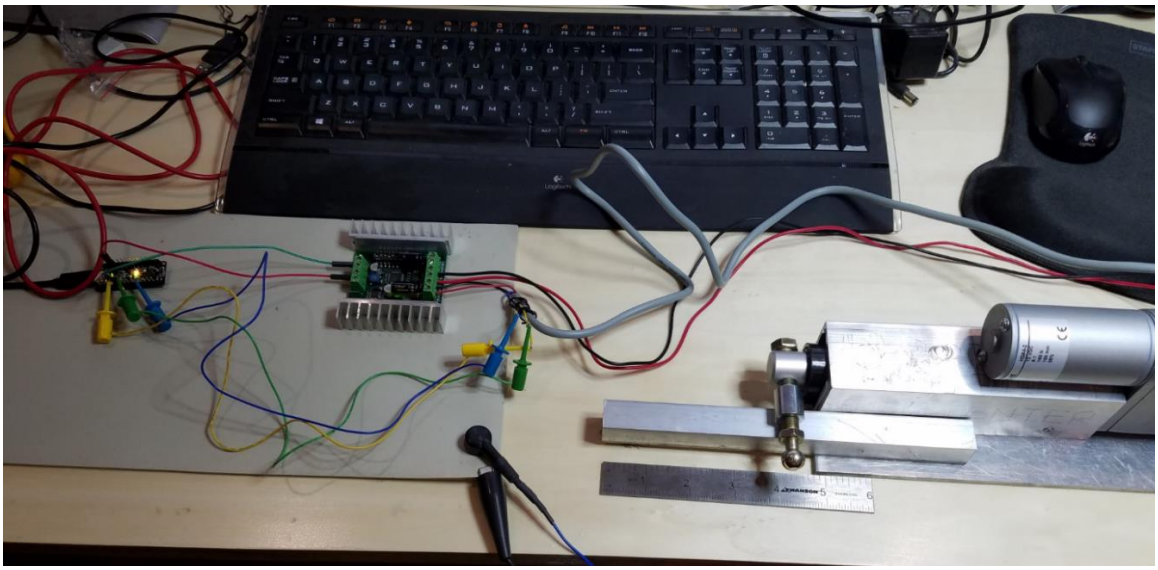


Figure 34. Bench testing an actuator

Once the brackets had been built and the actuators tested, they were then mounted to the frame of the mower and connected to the control rods. The final assembly can be seen in Figure 35. The advantage of this approach is that it easily allowed me to switch from manual to computer control simply by disconnecting the control arm linkages from the actuators and reconnecting them to the control arms. Later, absolute position encoders were added to the control arms to allow the computer to know where the operator had positioned the arms to move the linear actuators to the appropriate position, thereby allowing the mower to be controlled as it was originally intended.

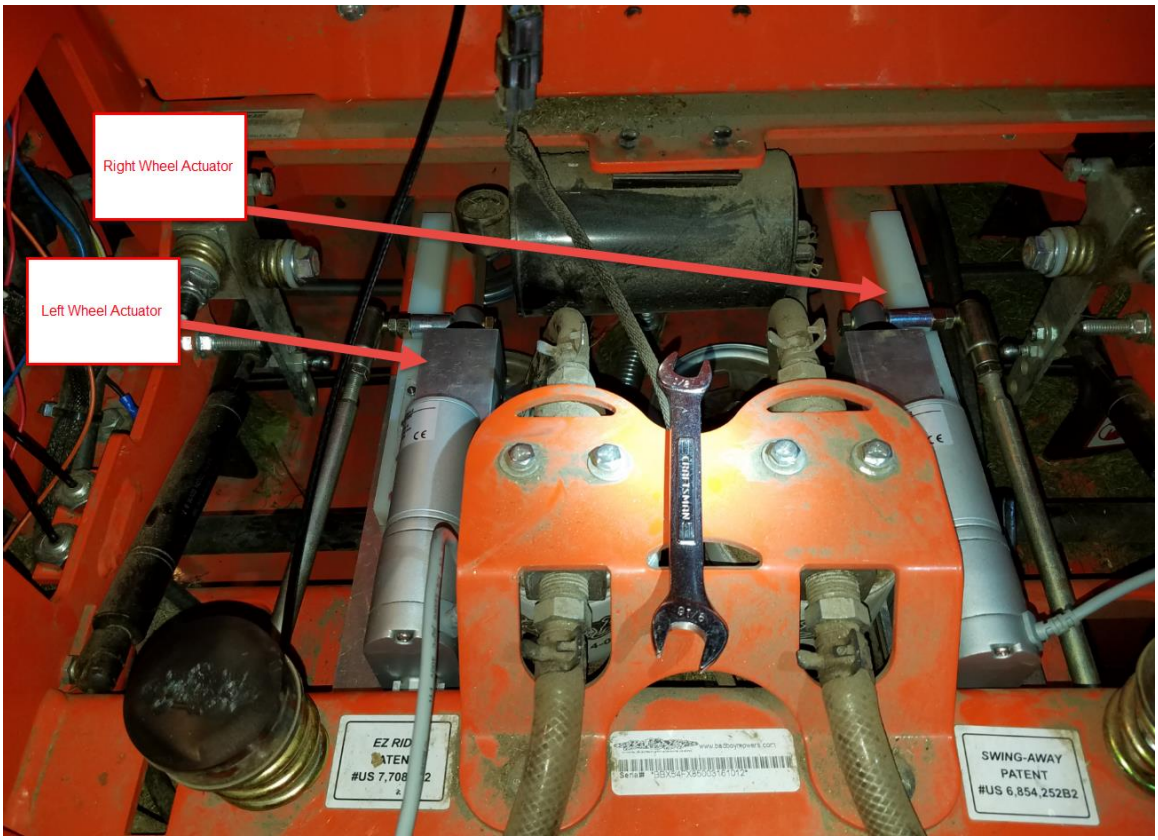


Figure 35. Mounted Drive Actuators

This approach had the added benefit of allowing the computer to record the operator's movements throughout a training session, which was used in conjunction with the GPS to define work-area boundary lines.

The linear actuators chosen to control the drive motors have a 4" stroke, a max speed of 2" per second and a dynamic thrust rating of 25 lbs. They also have a potentiometer for position feedback. The actuators were mounted to the frame so that the control arms rest in their neutral position when the actuators sit at the 2" mark. This means the actuators can push the control arms 2" forward to their full 4" extended position and pull back the control arms 2" to their fully retracted position. This represents nearly the full travel available for the control arms.

The linear actuators are controlled using a 2x12 Sabertooth Motor Controller (Dimension Engineering, 2012). This motor controller can be seen on the left side of Figure 34. The Sabertooth is capable of controlling 2 DC motors up to 12 Amps. The linear actuators are rated for a maximum load current drain of 3.8 Amps and a stall current of 15 Amps. I placed 10 Amp fuses in-line with the motor controllers to prevent overloading the Sabertooth in the unlikely event the 15 Amp stall current is ever reached. Control of the Sabertooth motor controller is discussed in Software Section 2.2.

2.1.3.2 Parking Brake (Linear Actuator)

The parking brake is an important part of the mower's operation. The mower is designed with a set of safety interlocks. The mower is not allowed to start the engine unless the parking brake is engaged. Additionally, the mower deck is not allowed to engage if the parking brake is on. Given that the mower is operated by hydraulic drive motors, it is able to maintain a locked position if the control arms are centered. This means that once the control arms are returned to their center position, as long as the mower's engine is running, the mower will hold its position, even on a hill. However, if the engine cuts out, the mower will start to roll as soon as the hydraulic fluid is no longer under pressure from the engine. When engaged, the parking brake prevents the mower from moving even if the engine is not running.

The parking brake was originally designed to be controlled through a hand-operated cam mechanism, as shown in Figure 36.

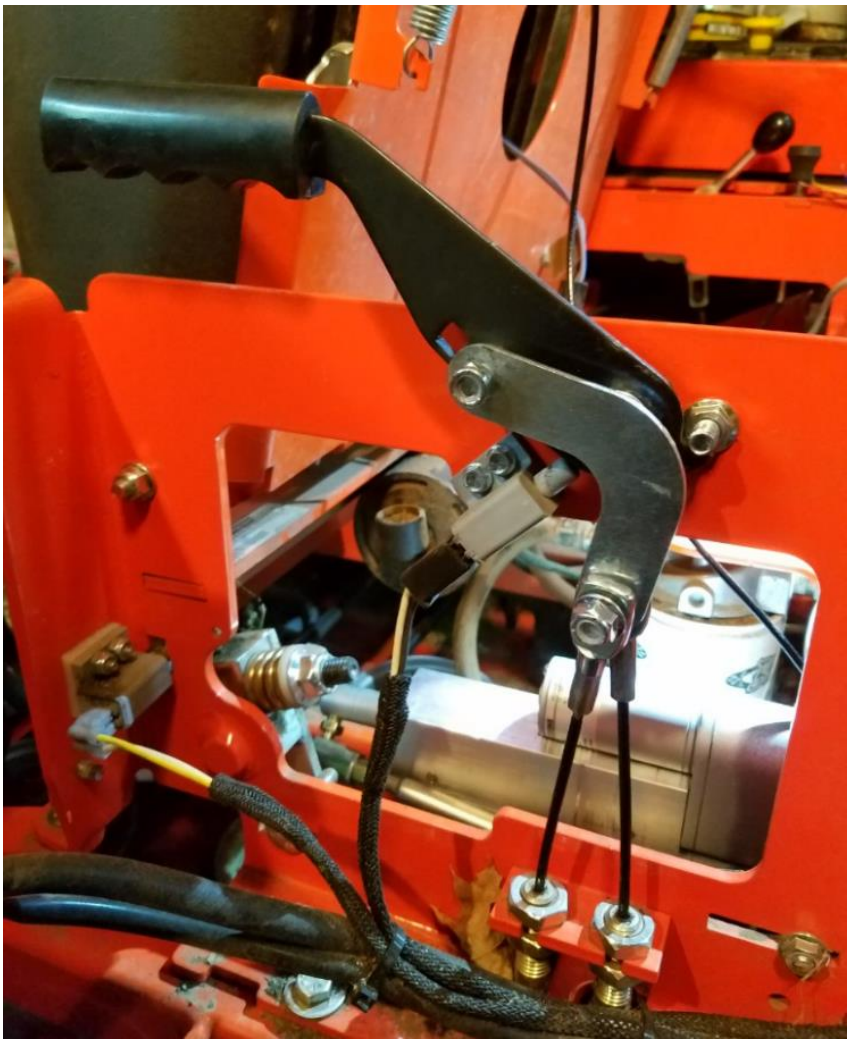


Figure 36. Unmodified Parking Brake

Pulling up on the handle caused two cables to be pulled taught. I was able to reuse the original cam bracket by flipping it around and creating a center pivot mount as shown in Figure 37. This approach allowed the parking brake to be engaged when the actuator rod was pulled in. Extending the actuator rod released tension on the cables, thereby disengaging the parking brake. Since these actuators operated using a lead screw, they hold their position once set, even if power is removed.

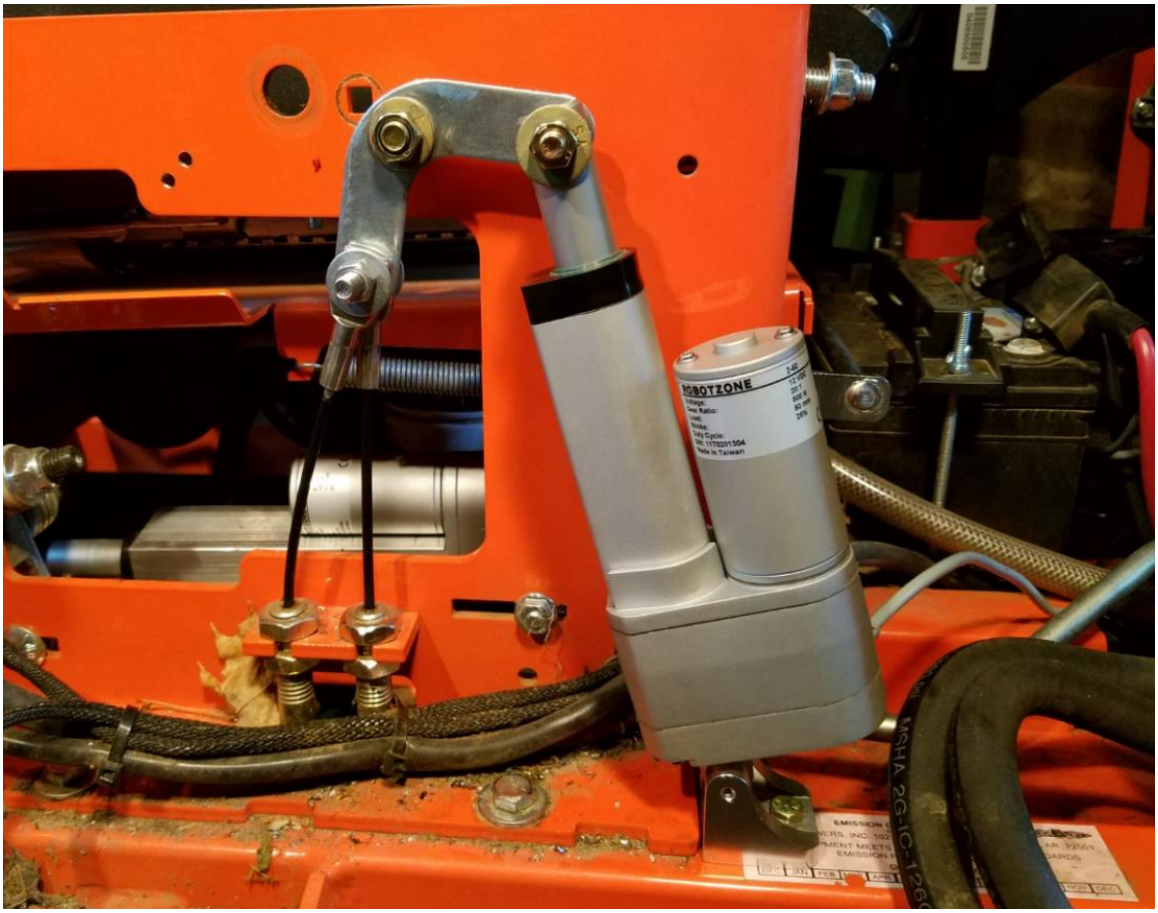


Figure 37. Parking Brake Actuator

Control of the parking brake actuator was initially intended to be done using a second Sabertooth motor controller. However, in order to more directly control the parking brake by the safety control mechanism, this was changed to relay control. Reasons for this are discussed below in Safety Protocol Section 2.1.9. Since the relay board chosen for use in the system only had Single-Pole-Single-Throw (SPST) relays, two relays were used to control the movement of the actuator.

2.1.3.3 Throttle (Servo)

The throttle in a mower controls the amount of power available to the system. The throttle controls the speed of the engine. The faster the engine, the more power is

available for use by the hydraulic motors and the Power Take-Off (PTO), which is used to run the mower deck. The amount of power required to move the mower around the yard is significantly less than the amount of power required to run the mower deck. The throttle must be increased to full power when running the mower deck. Therefore, control of the throttle was brought under computer control to allow for these types of dynamic changes in power requirements. Control of the throttle in the system is covered in detail in chapter 3.

2.1.3.4 Engine Start/Stop (Relay)

The electrical system in the mower is controlled through a series of interlocking relays. This means there is usually not a single relay that controls a particular operation, but instead multiple relays that must be in a given state before an operation is allowed to commence. For example, you cannot start the mower if you are not sitting on the seat with the control arms in the neutral position. There is a seat switch that engages when a person is sitting on the seat and this seat switch is connected to a relay. Likewise, the control arm switches are also connected to relays. Unless all relays are in their appropriate states, the system will not perform the expected operation.

My first thought as to how to tie the computer into these functions was simply to replace the mechanical switches with relays. However, given the hard-wired interconnected nature of the existing system, this proved unfeasible. It would have required multiple relays to replace individual functions. After careful analysis of the electrical system schematic (Figure 38), as well as taking electrical measurements on a meter to ensure I interpreted the schematic correctly, I decided to tie into the control

functions directly. This meant I was effectively bypassing the safety interlocks present in the system. However, as described under the safety protocol section below, these were reimplemented in software, with enhanced capabilities. Starting the engine required two relays. The first relay controlled the power on the wires to the engine that would normally prevent it from starting. The second was connected to the starter itself. Engaging the first relay is the equivalent of turning your car to the run position—it needs to remain in this position in order to continue running. The second relay is the equivalent of turning your car's ignition to the start position—doing so causes the starter motor to turn the engine, thereby starting the engine. This relay is engaged only until the engine is running, as determined by the tachometer in the throttle control system. The tachometer and throttle control system are described in chapter 3. Disengaging the first relay causes the engine to stop running and shut off. It is the equivalent of turning your car's ignition to the off position.

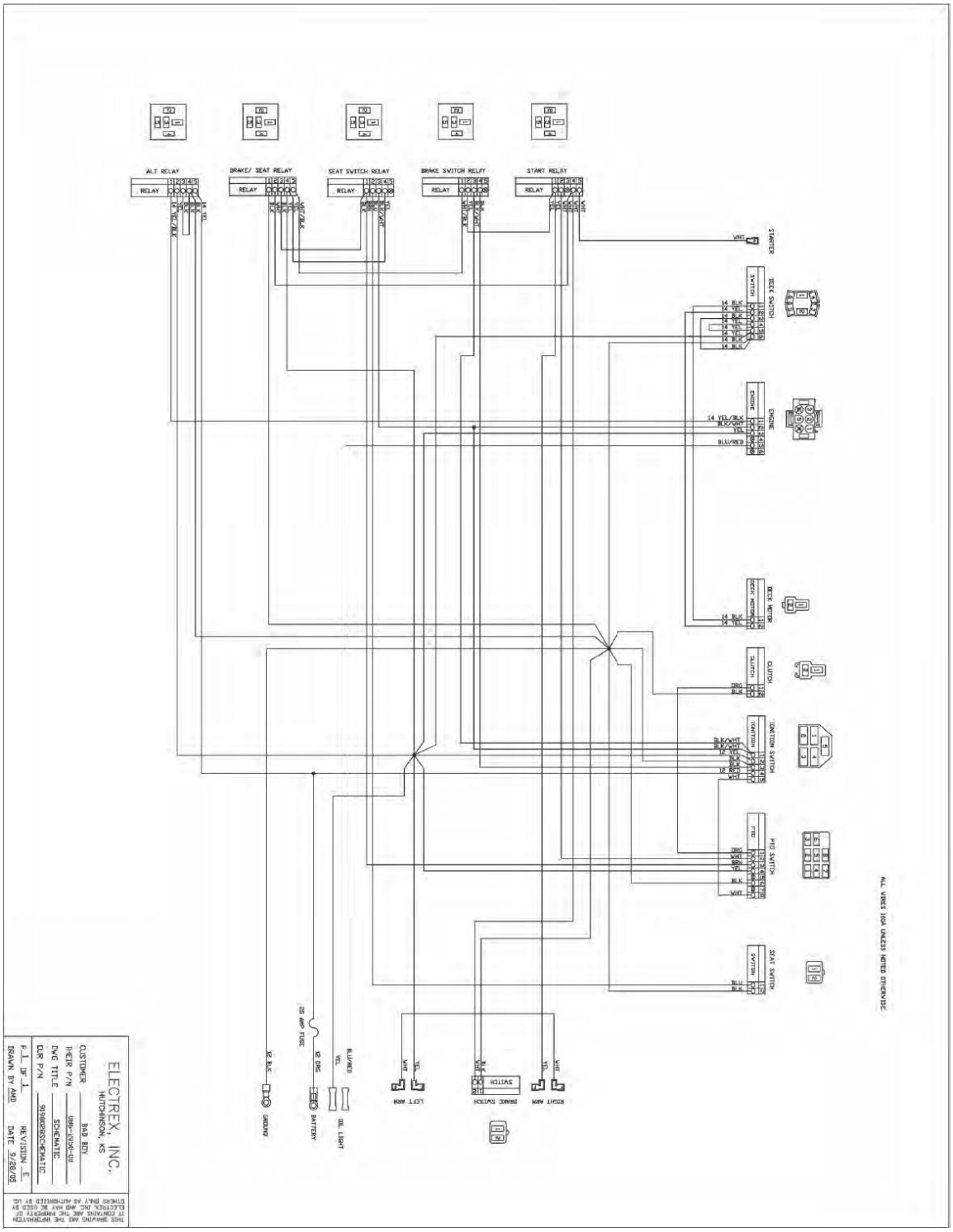


Figure 38. Outlaw XP Electrical Schematic (Bad Boy Outlaw XP Owner's Manual)

2.1.3.5 PTO (Relay)

The Power Take-Off (PTO) is what controls power to the mower deck. There is a manual push-pull switch that connects to a relay to allow the mower deck to engage and disengage. The mower deck will not be allowed to engage if the parking brake is engaged. Engaging the parking brake while the mower deck is running causes the engine to stop. I added a relay to the output of the PTO to act as if the PTO switch has been engaged when the relay is active and disengaged with the relay is inactive. The relay is controlled through a digital output pin on a microcontroller. Further details can be found in chapter 6.

2.1.3.6 Safety Cutoff (Relay)

As previously mentioned, there are many safety interlocks present in the unmodified mower system. These are designed to ensure the safety of the operator and bypassing them is not recommended. There is a need to be able to kill the mower's engine quickly. The section on the remote control details the need to shut down the mower if a radio signal is lost. In addition, a manual kill switch has been added to the mower to allow the mower to be quickly shutdown if the need arises. The question was how to tie into the system to allow the mower to be quickly shut down.

There is a seat safety switch on the mower that detects the presence of a human operator. If no operator is sitting in the seat, the mower will not start. If an operator gets up from the seat while the mower is running, the mower shuts down. Since the mower is meant to operate autonomously, this safety mechanism had to be bypassed. I added a relay to the system that engages and disengages the seat safety switch. This allowed me

to bypass the safety mechanism when the mower was operated in autonomous mode. It also provided a quick and easy way to kill the mower if the need arose—if the mower was running in autonomous mode and the relay disengaged, the mower would automatically shut down.

2.1.4 Sensors

There are a variety of sensors used in the ASMO system to provide the on-board computer with knowledge of its environment. The various sensor types are listed in table 3.

Table 3. Sensor Types

Sensor	**Description**	**Main Computer Connection**
GPS	Provides location information to 5-10 feet. Testing for accuracy will need to be done. The GPS will be used to loosely orient ASMO in the yard. One GPS used.	USB through a microcontroller
LIDAR	Custom-built laser guidance system	USB through a microcontroller (x3)
Hall-Effect	Magnetic sensor used to determine engine speed	USB through a microcontroller
IMU	Inertial Measurement Unit, combines 3 axes of accelerometer data, 3 axes of gyroscopic data, and 3 axes of magnetic (compass) data.	USB through a microcontroller
Temperature	Used to measure the internal temperature of the on-board computer case.	USB through a microcontroller
Hall-Effect	Magnetic sensor used to determine wheel speed and direction.	USB through a microcontroller (x2)

2.1.5 Main Computer

All processing for the mowing operations takes place in an on-board computer. The main computer board is called an UP Board and is manufactured by AAeon (AAeon, 2018). This board can be sourced through Mouser, Digi-Key, or Avnet Electronics. The board itself is shown in Figure 39. The form factor follows that of the popular Raspberry Pi, but with the power of an x86 processor instead of the Arm processor present in the Raspberry Pi board. The UP Board has an operating range of 32-140 degrees Fahrenheit, which was an important consideration considering summer temperatures. The $160 version I purchased for this project came with 4 GB of DDR3-1600 memory and a 64 GB eMMC storage card. The processor is an Intel Atom x5-8350 Quad-Core Processor that operates up to 1.92 GHz. It also has 4 on-board USB ports, HDMI output, and an Ethernet jack. Additionally, the board has a 40-pin bus that includes support for General Purpose Input/Output (GPIO). The GPIO support was not used in this project.

Figure 39. UP Board Main Computer

This board was chosen over other boards because of its small form factor, higher temperature operating range, and faster processor. While ROS would run on a Raspberry Pi, it would not run as quickly or efficiently as it would on the Up Board. Additionally, the faster speed of the Intel Atom processor made working with Linux much easier as the user interface was more responsive when development had to take place locally on the board. Figure 40 shows the main computing unit, mounted in the case I fabricated. The main computer board is mounted underneath the microcontrollers to the right. The Sabertooth motor controllers are shown in the middle and left-side of the case. The overall size of the computer unit is 10” x 6” x 2”—quite compact.

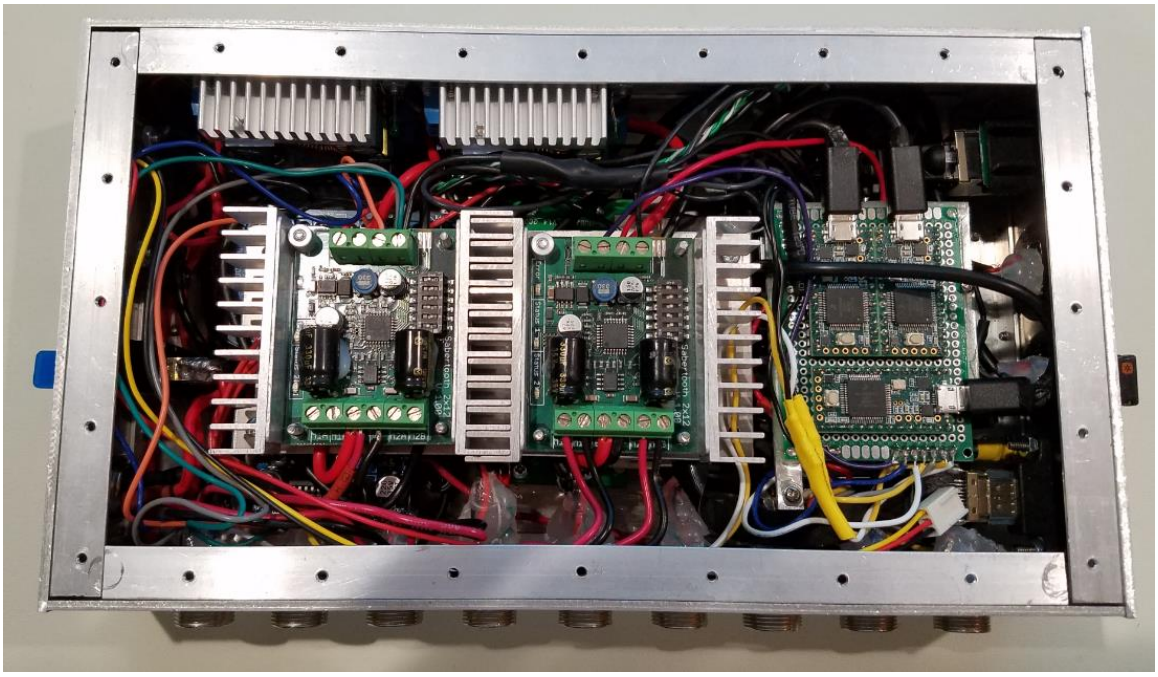


Figure 40. Main Computer Unit

2.1.6 Microcontroller

The microcontrollers used in this project are a combination of Adafruit Metro Mini 328 boards (Adafruit, 2018a) and Teensy 3.2 boards (PJRC, 2018). The Adafruit

Metro (Figure 41) contains an Atmega328 processor that runs at a clock speed of 16 MHz and is powered off 5 VDC. These microcontrollers can be programmed using the popular Arduino development environment. These microcontrollers are used in the LIDAR units. They are low-cost ($12.50 each) and can be powered directly off the USB port. Additionally, they can be programmed over the same USB port that is available for use as a Serial port after programming. This means these boards can be mounted inside the LIDAR units and re-programmed any time without the need to be removed. They support one hardware interrupt pin, which is used by the homing mechanism in the LIDAR units. The homing mechanism is used by the LIDAR units to know when the laser is facing a pre-specified location. More details on these microcontrollers can be found in chapter 4 on the Laser Distance System.

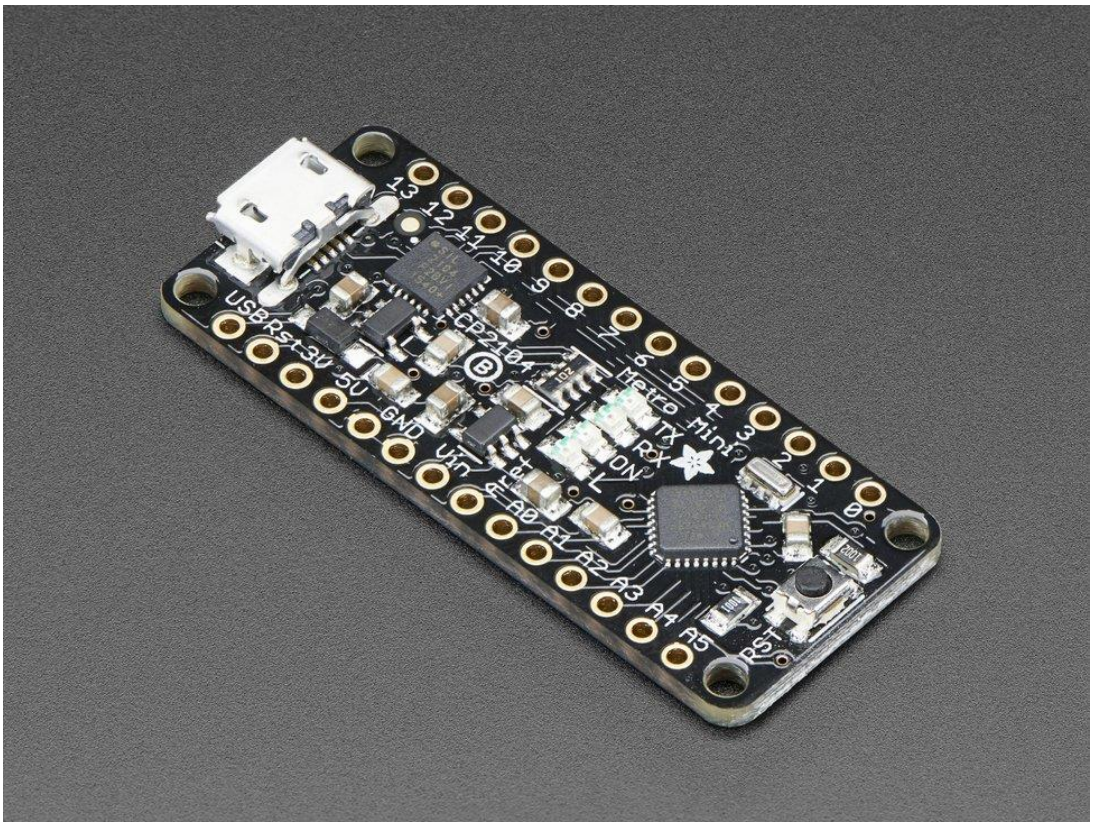


Figure 41. Adafruit Metro Mini 328

The Teensy 3.2 boards are more powerful than the Metro Mini boards and, as such, they cost more as well ($19.95). These boards are powered by a 32-bit ARM Cortex-M4 running at 72 MHz. An important reason for choosing them in this project is that they have 256K of Flash Memory and 64K of RAM. They also have the added benefit of being able to utilize any of their 34 digital pins as interrupt pins. This makes them much easier to interface to ROS. The Teensy boards are used inside the main computing unit for the watchdog system (section 2.1.7), the throttle control system (chapter 3), and wheel encoders (chapter 4).

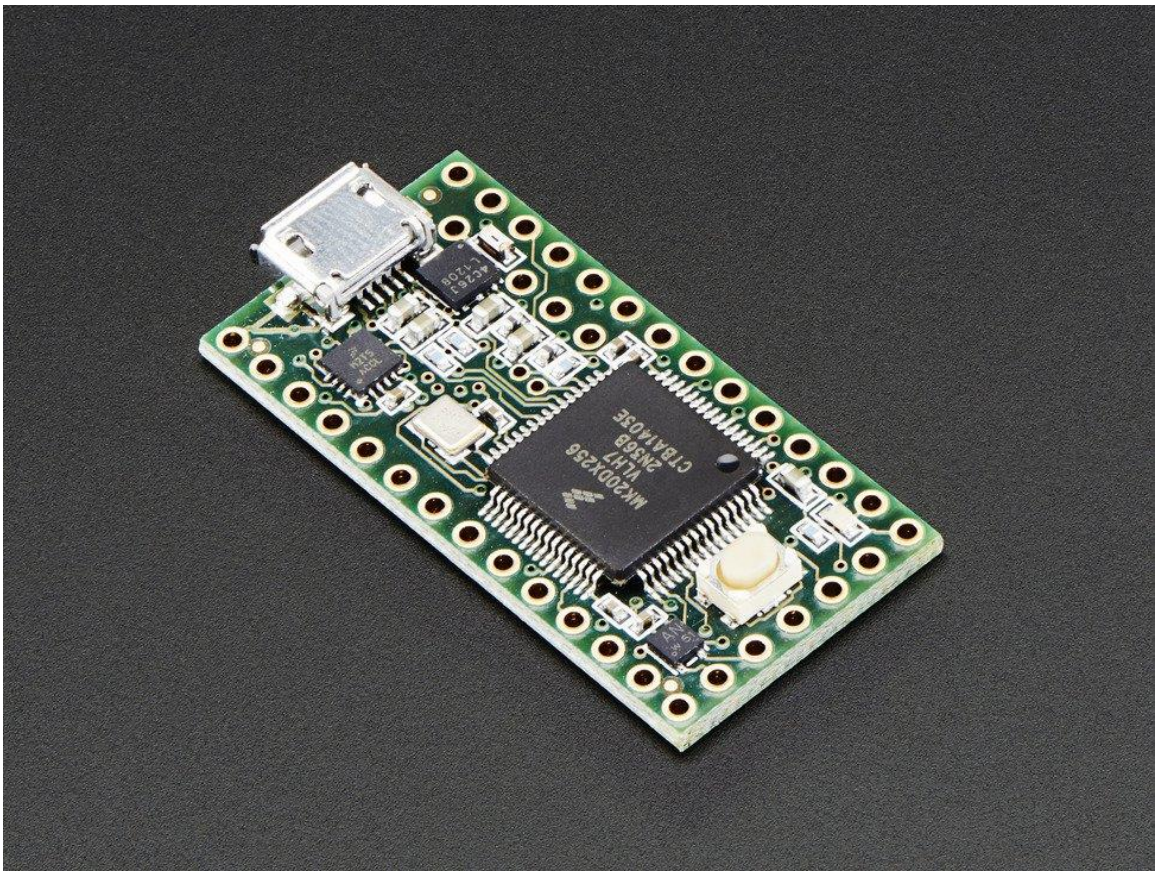


Figure 42. Teensy 3.2, designed by Paul Stoffregen and PJRC

Lastly, there are microcontrollers used for radio communications from Low Power Labs, which are based on the Atmega series of chips and called Moteinos (Low Power Lab,

2018). There are two types of Moteino microcontrollers utilized in ASMO. The first is the base unit Moteino and is used to receive radio signals from the second, remote, Moteino. Both operate at 933 MHz and have a range of 1,000 feet. The Moteino is capable of bit rates from 1.2 kbps to 300 kbps, but defaults to 55.555 kbps. The base unit Moteino MEGA (Figure 43) is based on the Atmega1284 chip while the remote unit Moteino (Figure 44) is based on the Atmega328 chip. The base unit Moteino is also used to interface the GPS and IMU units as well as a relay board for controlling the warning safety light on the mower.

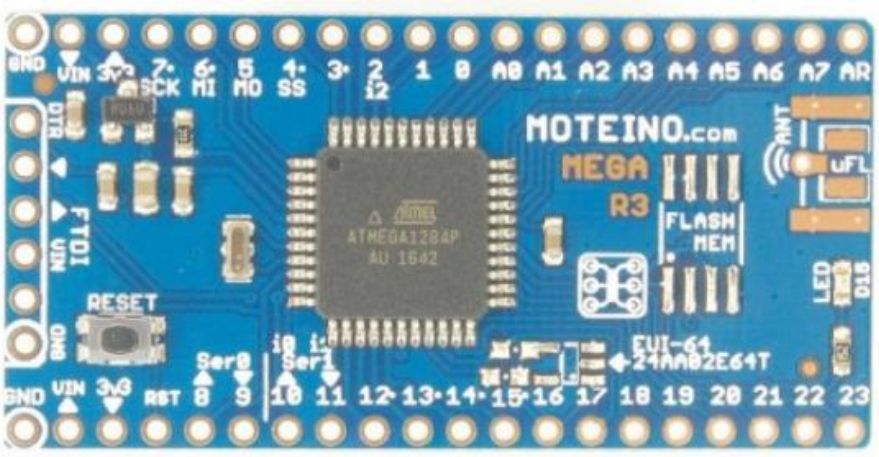


Figure 43. Base Unit Moteino MEGA

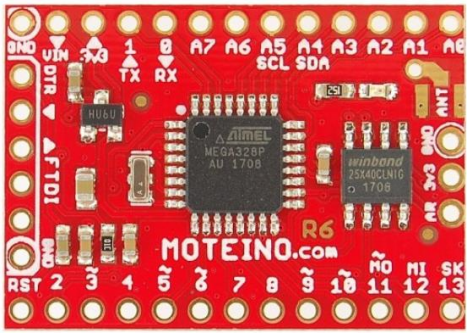


Figure 44. Remote Control Unit Moteino

2.1.7 Watchdog System

The watchdog system monitors the health of the main computer. This was done to avoid the mower's continued operation in the event of an unexpected main computer shutdown or lockup. The watchdog system is built using a Teensy 3.2 microcontroller and has a higher operating temperature range than the main computer. The Teensy 3.2 has an operating temperature range is from -40 to 185 degrees Fahrenheit. It has been mounted inside the same case as the main computer and is connected to the main computer through a USB port for communication. The watchdog monitors the internal temperature of the case to ensure it does not reach a critical limit. In addition, it has control over two small fans mounted inside the main computer case that it can use to cool the main computer down when needed. In my experience, computer systems tend to either shutdown unexpectedly or stop functioning when they reach an over-temperature condition. The watchdog continually pings the main computer through the USB port to ensure the main computer remains responsive. If at any point the main computer stops responding, the watchdog shuts down the mower by disengaging the safety seat relay. This failsafe process was put in place to ensure the main computer doesn't issue a command to the mower to move forward and start mowing at a particular rate and then stop functioning.

2.1.8 Remote Control

A remote-control unit (Figure 45) has been created that allows the mower to be started, stopped, and driven around without having to sit on the mower directly or utilize a remote session with the main computer. The remote-control unit utilizes a Moteino from Low Power Labs to read the switch inputs utilizing digital input pins. The joystick

is comprised of three potentiometers: the X-axis, the Y-axis, and a twisting motion. The Moteino microcontroller is also used to control the output to an OLED display. The remote control has a 1,000-foot range and works through buildings and other obstructions.

Figure 45. ASMO Remote Control

The microcontroller translates the X and Y axis on the joystick to Turn Power and Drive Power values that it then transmits to the base unit. The microcontroller reads the X and Y potentiometers from the joystick using 10-bit analog-to-digital inputs, translating the position of the X and Y axes to values between 0 and 1023. In my configuration, 0 on the Y axis means the joystick is pulled all the way down while 1023 on the Y axis means the joystick is pushed all the way up. Likewise, 0 on the X axis means the joystick is pushed all the way to the left while 1023 on the X axis means the joystick is pushed all the way to the right. The 0-1023 X value is translated to a Turn Power value from 0-127. The 0-1023 Y value is translated to a Drive Power value from 0-127. These mapped values are then sent over the radio to the base unit. The values are mapped to smaller numbers for two reasons. First, it is less data to transmit over the

radio—which transmits at 55.555 kbps—and second, the Sabertooth motor controllers also only use 7 bits for controlling the Pulse Width Modulation (PWM) outputs for the motors. The software interface is covered in more detail below, in section 2.2.

2.1.9 Safety Protocols

Safety is a primary concern in this project. As such, several safety protocols were implemented in order to provide safe autonomous operation of the mower. The safety protocols involve both hardware and software and are discussed in detail below.

2.1.9.1 System Interlocks (Seat, PTO, Engine Start)

The Bad Boy Outlaw XP mower comes equipped with a set of hardware safety interlocks to prevent the system from operating in an unsafe mode. These include the inability to start the engine if the parking brake is not applied, the inability to start the mower deck if the parking brake is applied, and the inability to run the engine if an operator is not present on the seat. While the original safety systems were bypassed in order to make the mower autonomous, their functions were kept intact and implemented in software. This means the engine will still not start unless the parking brake is applied and the mower deck will still not enable unless the parking brake is not applied. Additionally, when the mower is operating in autonomous mode, a flashing yellow warning light turns on. The required presence of a human operator on the mower was replaced with a wireless heartbeat.

2.1.9.2 Network/Wireless Heartbeats

As a safety protocol, in order for the system to operate autonomously, I programmed in a heartbeat function. This means the system must constantly receive a wireless heartbeat from the remote-control unit. This means the remote-control unit must be on and transmitting a signal to the mower. The remote-control unit transmits a signal whenever new data is present. For instance, if the joystick position changes or a button press is detected, these events are transmitted immediately. In addition, if there is no new data to transmit, the system transmits a heartbeat signal every 500 milliseconds. If the base unit misses two of these transmissions in a row, it shuts the mower engine off and applies the parking brake. This means if the remote-control unit is shut off, 1 second later the mower shuts down. It also means if the mower traveled outside the range of the remote-control unit, the mower would shut down. This safety feature is similar to that used in radio-controlled (R/C) aircraft.

The base unit radio receiver has direct control over the engine kill switch and the parking brake actuator. This allows the radio receiver to both kill the engine and apply the parking brake without the need to rely on the presence of the main board computer. Since the main computer is not a real-time system, there would be no guarantee that it would be running and available to service a shutdown request in real-time.

2.1.9.3 GPS Garden

There is a GPS present on the ASMO system that is used to track the mower's position on the property. The GPS is accurate to approximately 9 feet. In order for the mower to operate autonomously, it is required to have a GPS signal lock, which means it

has position information. When training the mower in a new operating area (detailed in the User Interface section), the GPS coordinates of the operating area are recorded. I wrote a custom ROS service to constantly monitor the output of the GPS. This GPS Garden Service sends a shutdown message to the base unit if it detects the mower passing outside the work area while operating in autonomous mode. This prevents the mower from wandering beyond the operator defined work area.

2.1.9.4 Kill Switch

A large red kill switch was mounted on the mower to allow an operator to easily stop the mower by simply slapping the large red toggle (Figure 46). This button is connected between power and the main computer. Pressing the stop button causes power to the main computer to be lost. This causes the system to immediately go into a default state, which means all relays open. I set the system up in such a way that when the relays open, the parking brake actuator is automatically applied and the operator presence relay opens, which tells the mower an operator is not present and so it shuts down. The relay board is powered directly by the mower's battery and does not pass through this switch.

Figure 46. Emergency Stop Button

2.1.9.5 Thread Priority Scheduling

One aspect of the ROS software is the ability to control thread priority for the various services that are running. ROS has a number of priority levels that its services can run under (Table 4). By default, the ROS services in this environment run at NormalPriority. The safety services built for this system run at RealTimePriority1 to ensure they have the highest possible priority in the system. This includes the GPS Garden Service and the Watchdog Service.

Table 4. ROS Scheduling Priority Level

BackgroundPriority (lowest priority)
LowPriority
NormalPriority
HighPriority
CriticalPriority

RealTimePriority4
RealTimePriority3
RealTimePriority2
RealTimePriority1 (highest priority)

Changing the Scheduling Priority Level in ROS changes the priority level of the Linux process running the service. Since the system is not running real time Linux, threads running under higher priority no longer run under the normal user space scheduling manager. Instead, the real-time process thread is guaranteed to get CPU time whenever the thread needs it. All other threads in the system must wait until any real-time threads are waiting or sleeping. For this reason, the GPS Garden Service and the Watchdog Service were written to perform their tasks as quickly as possible, much like an interrupt service handler. This was done to prevent these services from starving the rest of the system of CPU time.

2.2 Description of the Software

There are two main components to the software system. The first is a lower-level sensor, mechanical control, and remote-control interface. The second is a higher-level system, responsible for the autonomous behavior of the mower.

2.2.1 Lower-Level System (Microcontroller)

The lower-level system uses multiple microcontrollers to interface the various sensors in the system that do not have a direct interface to the main computer. For example, the IMU uses an I2C interface. The main computer has no built-in support for

I2C, but this interface is typically found on microcontrollers. Additionally, the main computer has no direct way to control the relays or linear actuators used to mechanize the mower. Microcontrollers provide an ideal way to bridge this gap.

Microcontrollers have digital output systems ideal for controlling relays. The relays are responsible for operations such as starting the engine and turning the cutting blades PTO on and off. Microcontrollers also have outputs that can be used to drive circuits for PWM control of motors such as linear actuators, which are used to control throttle speed on the mower. The microcontrollers have a simple USB interface that can be used to connect to the main computer. There are four types of microcontrollers used in this system: 1) Adafruit Metro, 2) Low Power Lab Moteino MEGA, 3) Low Power Lab Moteino, and 4) PJRC Teensy 3.2. There were various types of programming techniques used to allow the microcontrollers to interface to the main computer unit. Additionally, while several of the sensors used in this project had software libraries available for communicating with them, such as the IMU from Adafruit, others had to be created from scratch.

An advantage to using the Teensy 3.2 and Moteino MEGA boards is that they have enough memory to run the ROS Serial library for the Arduino. Since both the Teensy and the Moteino are capable of being programmed with the Ardunio environment, this proved ideal. The ROS Serial library implements the rosserial protocol, which is a standard way in ROS for serializing and multiplexing messages over character devices such as serial ports or network sockets. This package library made it easy to expose microcontroller-based functionality to the rest of the ROS system. ROS is covered below in section 2.2.2.

The Adafruit Metro and Moteino (regular, non-MEGA version) do not have enough memory to support running the ROS Serial library. The regular Moteino was used in the remote-control unit. I created the message structure and serialized it manually for use by the remote-control base unit. This base unit runs on the Moteino MEGA, which does run the ROS Serial library. This means that any messages not directly consumed by the base unit can be re-emitted into the ROS system using the ROS Serial library running on the Moteino MEGA.

The Adafruit Metro microcontrollers run on the LIDAR units. These boards do not have enough memory to run the ROS Serial library. Instead of leveraging this library to inject ROS laser messages into the rest of the ROS system, I wrote a custom Linux-based ROS service in C that receives the serialized messages from the Adafruit Metros and re-publishes them as ROS laser messages to the rest of the ROS system. While this approach required more work, it meant that the LIDAR units could be used in other, non-ROS environments, such as Microsoft Windows. ROS does not currently support Windows. The LIDAR units are covered in detail in chapter 4.

2.2.1.1 PID for Throttle Governor

A Proportional Integral Derivative controller algorithm (Beauregard, 2011) was implemented to control the throttle plate. This software governor system is covered in detail in chapter 3.

2.2.1.2 Relays, PWM for Hardware Control

Many of the functions on the mower are controlled using relays. Starting and stopping the engine, turning on the safety warning light, activating and deactivating the seat switch to kill the engine, interacting with the parking brake, and turning on and off the mower deck are all controlled using relays. The relay boards used in this project are from JBtek (Figure 47).

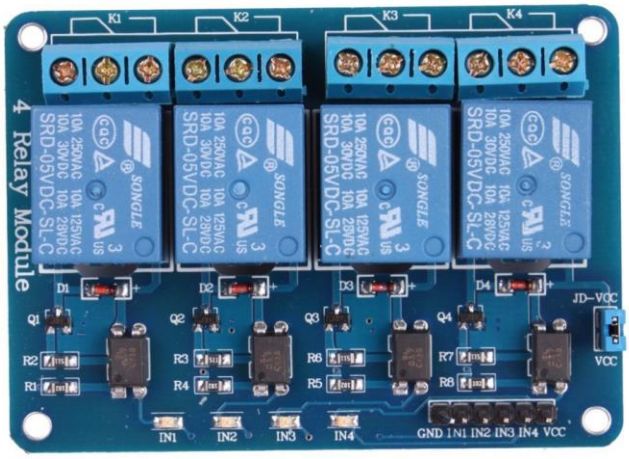


Figure 47. JBtek relay board

Finding low-cost 3.3 VDC relay boards is next to impossible. Most relays are driven by at least 5 VDC. The JBtek 4 relay board (Amazon, 2018a) costs $6.99 from Amazon. One advantage of the JBtek relay boards is that the relays are enabled when pulled low. This means that while they are 5 VDC relays, they can be driven by the 3.3 VDC microcontrollers used in this project as ground is common between the relay board and the microcontrollers. Two such boards were utilized in this project. One was mounted in the radio base unit and the other in the main computer unit (Figure 48). The radio base unit relay board controls the warning light, the seat kill switch, and the parking

brake. The parking brake required two relays to operate since the relays were SPST and there were two lines to be controlled (positive and negative power). The main computer unit relay board controls the engine start/stop and the PTO for the mower deck.

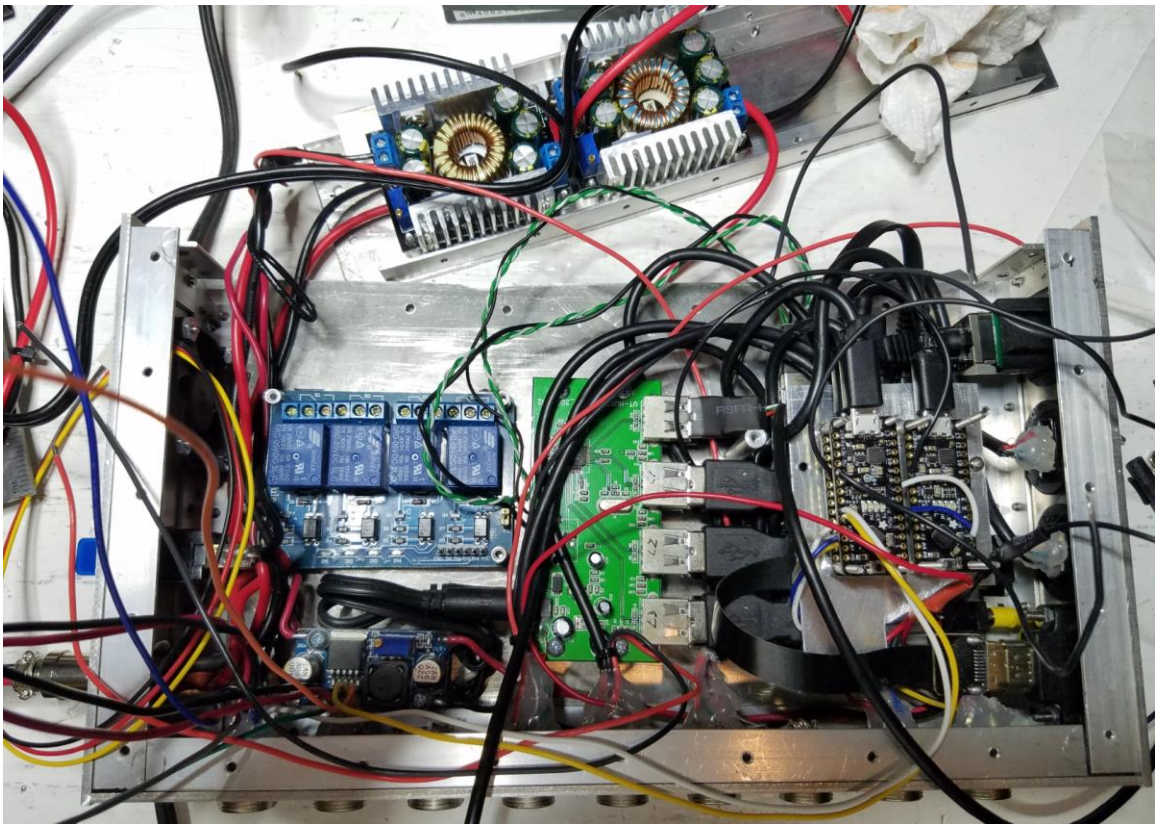


Figure 48. Dismantled main computer case, exposing relay board

Pulse-Width-Modulation (PWM) is used for controlling the throttle servo and the drive motor linear actuators. Control of the throttle servo is covered in chapter 3. The drive motor linear actuators are controlled using PWM through a Sabertooth motor controller (left/center in Figures 40 and 49). The Sabertooth unit is controlled through a simple serial protocol as documented by Dimension Engineering. This protocol uses TTL level single-byte serial commands to control the motor speed and direction. Only

the transmit line from the host system is needed to communicate with the Sabertooth in this mode. The Sabertooth uses a single byte to control both motors.

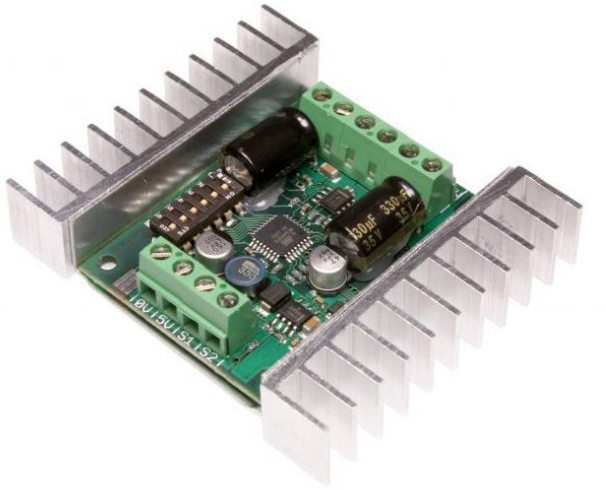


Figure 49. Sabertooth 2x12 Motor Controller

The 7 lower bits of the byte indicate the motor speed and direction while the upper-most bit indicates the motor the command is intended for. As such, sending a value from 1-127 controls motor 1 while a value from 128-255 controls motor 2. For motor 1, the value of 1 is full reverse, 64 is stop, and 127 is full forward. For motor 2, the value of 128 is full reverse, 192 is stop, and 255 is full forward. The value of 0 is "all stop" for both motors. A custom ROS service was created to take movement commands, from a planner module or the remote-control unit, and translate the movement commands into Sabertooth commands.

2.2.1.3 I2C, Analog, Serial for Environmental Monitoring

There are several environmental sensors used in the ASMO project. First, there is a precision temperature sensor from Adafruit shown in Figure 50 (Adafruit, 2018b). This

sensor is inexpensive ($4.95) and accurate (+- 0.25 degrees Celsius). This sensor is mounted inside the main computer case and is connected to a Teensy 3.2 microcontroller board through I2C. Adafruit provides a simple library for reading the temperature from this device. The ROS Watchdog service I created monitors this sensor and outputs ROS messages with its readings.

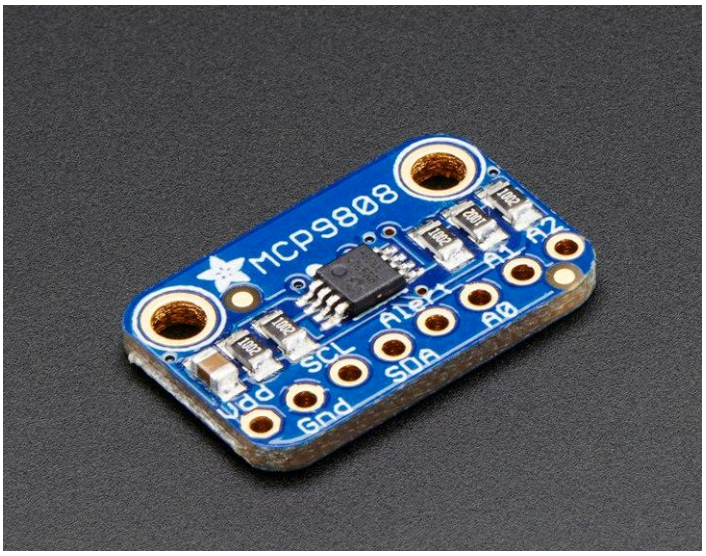


Figure 50. Adafruit MCP9808 breakout board (Adafruit, 2018b)

The Adafruit Ultimate GPS breakout board (Adafruit, 2018c) was used to provide GPS coordinates in the ASMO project. This board is low-cost ($39.95) and provides fast (10 Hz) updates. It has support for 66 channels and can run on 3.3 VDC. It also has a battery to store ephemeris data from satellites for faster warm-starts. It has a cold startup speed of 34 seconds. I am reading the raw NMEA sentences (Raymond, 2016) outputted from this device over a serial line on the Moteino MEGA. These messages are then being parsed and emitted as a GPS message to the rest of the ROS environment.

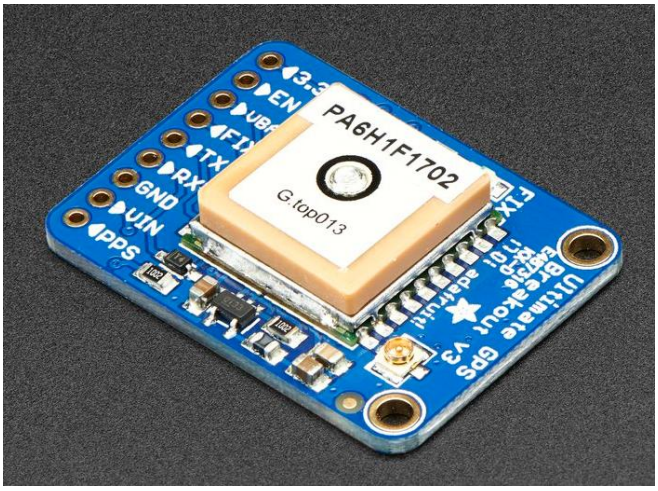


Figure 51. Adafruit Ultimate GPS breakout board (Adafruit, 2018c)

The last environment sensor used in the ASMO project is an Inertial Management Unit (IMU). Adafruit is called on again for their Bosch BNO055 breakout board (Adafruit, 2018d) as pictured in Figure 52. This board blends accelerometer, magnetometer, and gyroscope data to provide stable three-axis orientation output. Adafruit provides a software library for the Arduino environment to read the data output from this breakout board. I am using the Moteino MEGA microcontroller to read the data from this board and emit it as a ROS message. The data from this sensor is then fused with data from the wheel encoders to provide more accurate odometer data. Details covering the implementation of this sensor fusion and localization are covered in chapter 5.

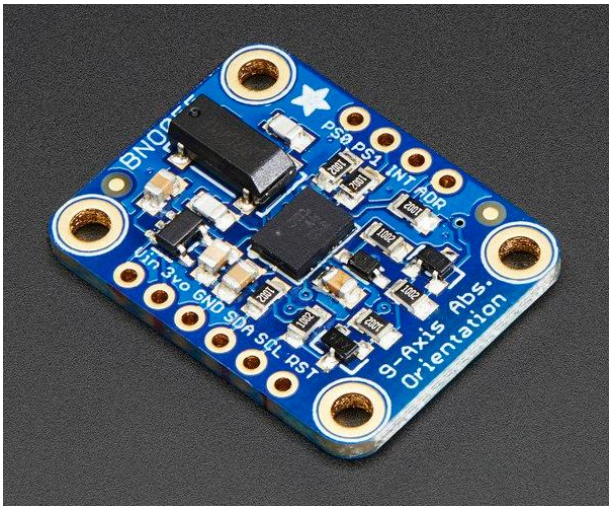


Figure 52. Adafruit 9-DOF Absolute Orientation Bosch BNO055 breakout board (Adafruit, 2018d)

2.2.2 Higher-Level System (ROS)

The higher-level system runs on the main computer under the Linux operating system. This system is based on the Robot Operating System (ROS). ROS can be thought of as a meta-operating system—an operating system running on top of an operating system. ROS is a framework for writing software for robots in a reusable, shareable way. ROS is comprised of multiple nodes running as individual software processes. The various nodes are loosely coupled together and use a message-passing interface for communication over sockets. Since these components are not tightly coupled, they can be developed and used outside of any particular implementation, so long as the messaging interface is respected.

This reusability is one of ROS' major benefits. It means that universities no longer have to reinvent the wheel every time they want to do development on some new aspect of robotics. Before ROS, if a university wanted to investigate ways of optimizing path planning, it meant they had to start from scratch and essentially create all the other

software aspects of the robot before they could start on their specific research. With ROS, existing components can be plugged together requiring development of only those modules specific to their current research.

The ASMO project drew on the work of other researchers through the openly available ROS modules. I also created many of my own ROS services—some due to the use of custom hardware and some due to the implementation of custom algorithms. I leveraged the built-in ROS services for handling map-building and localization, specifically the use of the gmapping Simultaneous Localization and Mapping (SLAM) service. Leveraging existing services where possible provided me with a starting point so I didn't have to begin from square one. My primary goal with this project was to create a system capable of mowing a complex lawn in an efficient manner. Leveraging existing services where possible allowed me to focus my time more effectively.

The modular, loosely coupled nature of ROS combine to form a system that can be developed and debugged offline, outside the hardware of the robot itself. The various ROS modules communicate with each other by passing messages between processes. So, when a module, such as the path planner, decides to move the robot forward for a period of time, it will send a message to the drive module. ROS has built-in tools designed to capture and play back these messages. This makes debugging robotic systems built with ROS much easier. ROS comes with a tool called RVIZ that provides for robot modeling and visualization. RVIZ is also capable of playing back captured messages.

RVIZ is a three-dimensional tool in ROS that is used for visualizing the robot as well as sensors and the environment. It allows for the quick identification of problems

related to both hardware (e.g., sensor misalignment) and software (e.g., that an algorithm is generating incorrect data). Figure 53 shows an example of the RVIZ environment.

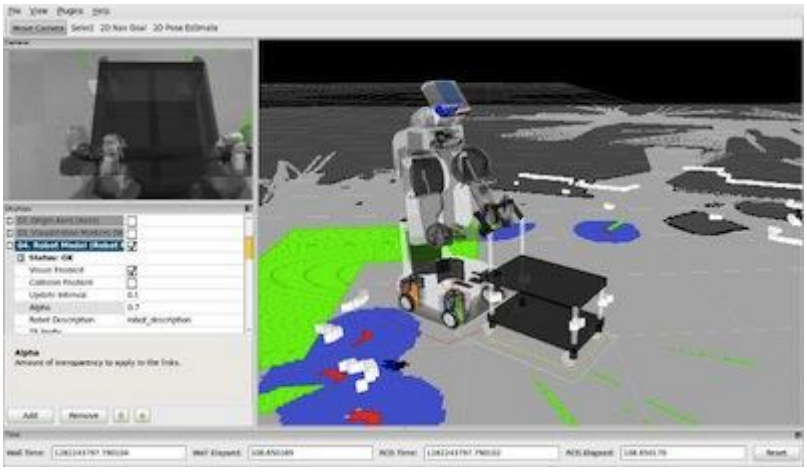


Figure 53. The ROS RVIZ program

In addition to the gmapping service, there were other services I was able to leverage in the ROS environment. These include the amcl (adaptive Monte Carlo localization) service and the navigation service. The amcl service is a probabilistic localization system for a robot moving in two dimensions. It implements the adaptive Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map. The navigation service is a two-dimensional navigation stack that takes information from odometry, sensor streams, and a desired position, outputting velocity commands to the robot base. The relationship of these various services can be seen in Figure 54.

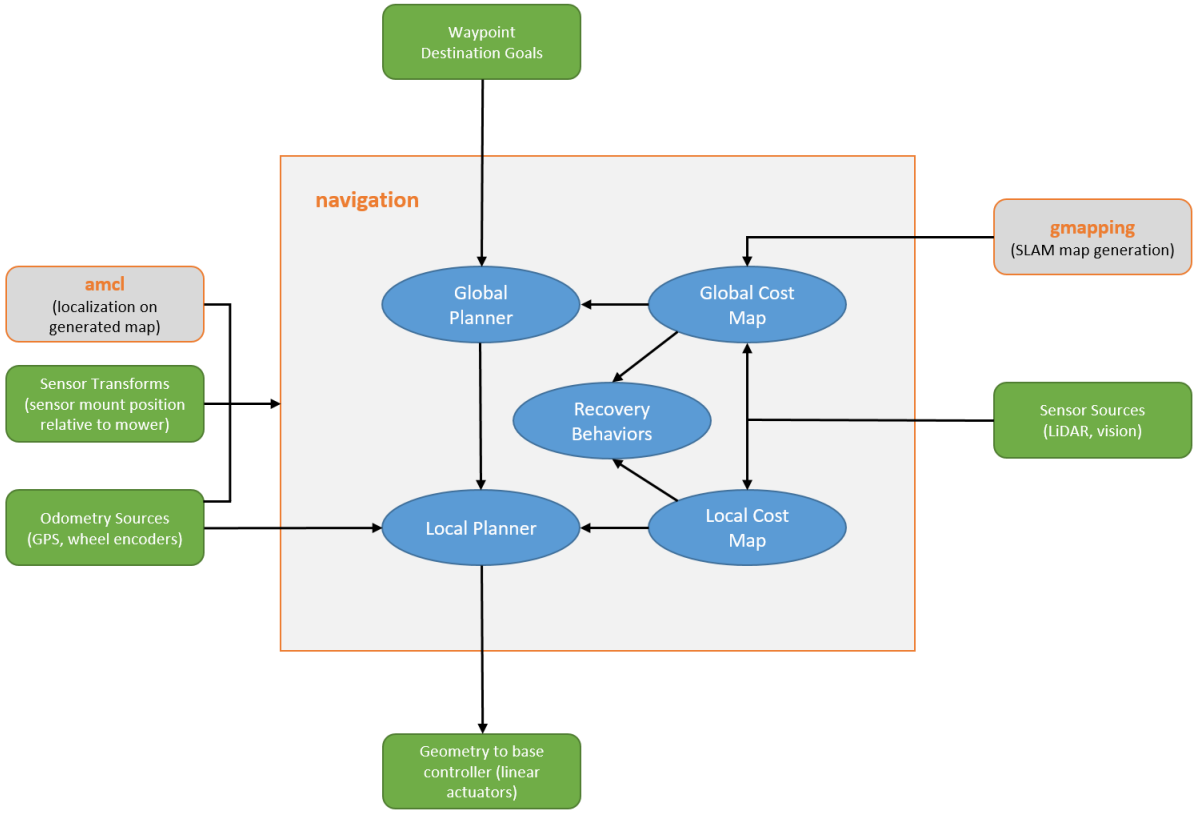


Figure 54. The ROS navigation system

While my original intent was to use as much of the built-in ROS navigation system as possible, as previously mentioned, I ended up implementing my own global planner, which is covered in detail in Chapter 7.

2.2.2.1 Component-based Software Services

The component-based nature of ROS is one of its strongest properties. The ability to develop and debug individual software components allows for a more focused approach to software development. Each component defines a set of messages it can publish along with a list of topics it wants to subscribe to. ROS provides a comprehensive list of message types that range from primitive types such as integers and floating-point numbers to more specialized messages such as GPS, robot pose, and sensor

messages. If the message structures are adhered to, existing components can be unplugged, and new components plugged in as needed. This allows the programmer to create new components that implement new algorithms, then, as long as the inputs and the outputs remain the same, the system can be restarted with these new components without requiring changes to the rest of the environment.

The throttle control unit discussed in chapter 3, the custom LIDAR units discussed in chapter 4, the fused odometer unit discussed in chapter 5, the user interface units discussed in chapter 6, as well as the custom global planner discussed in chapter 7 all implement custom ROS services that allow the specialized hardware and software to plug into the rest of the ROS environment. As an example, there are already existing laser services for a variety of the more expensive LIDAR units available today. These are the ROS services that currently work with the ROS navigational system. When I created my own custom hardware, I implemented a ROS service to expose my laser data to ROS as a normal laser-based ranging sensor. This allowed the rest of the ROS environment to operate without knowing the source of the laser data.

Another built-in feature of ROS is its distributed parameter system. The distributed parameter system is a way of exposing shared configuration information between services. It is implemented as a global key-value store. This approach allowed for the easy customization of ROS components. For instance, I used this mechanism to setup the LIDAR services I implemented so that the various ports used by the LIDAR units could be pre-configured. This meant I could define the front-left LIDAR unit as always existing on ttyACM0 while the front-right LIDAR unit exists on ttyACM1. Another feature of the ROS configuration environment is the ability to remap output

messages from services. I used this approach so that the one ROS service I had for communicating with my custom LIDAR units emitted a LaserMessage. This LaserMessage output message could then be remapped to FrontLeftLaserMessage while a second instance of the same service could have its LaserMessage output as FrontRightLaserMessage.

This black box approach to software development enforces clear lines between various components. This improves encapsulation and promotes code reuse. This component-based approach is a powerful way of developing software.

2.2.2.2 Event-based Communication

One form of communication in ROS is through the use of message-passing. This form of communication is called Topics. The messages the various ROS services pass to each other is handled through an anonymous publish and subscribe system. This system is asynchronous, meaning there is no waiting for a response to a particular message. A service emits a message and then carries on with its tasks. Any other services present in the environment can subscribe to these messages. Some messages can be emitted at periodic intervals, such as GPS data arriving 10 times per second, while other messages may only emit when data is available, such as with the remote-control unit.

2.2.2.3 RPC Communication

Remote Procedure Calls (RPC) communication in ROS is achieved through the use of Services. A ROS Service is a synchronous form of communication where the caller needs to wait for a response before continuing. An example of this form of

communication is a service that needs to add two numbers together before continuing. ROS also supports preemptable RPC calls. This form of communication is through the use of a ROS Action. Actions are RPC calls that can report back on progress before the final result is returned. Preemption means the caller can cancel or otherwise alter the Action as it progresses. An example of an Action is having a robot navigate a series of waypoints to a destination, but then the caller decides to alter the course during the trip due to additional sensor data that has been obtained or other commands that have been received.

2.2.2.4 ROS Visualizer/Debugger

The component-based message-passing nature of ROS enables easy debugging of complex robotic systems. ROS provides tools to record messages as they are passed around the system. These messages can then be played back later for further analysis. This process has been very helpful during the creation of the sensor fusion system. Sensor data messages from the IMU and wheel encoders could be recorded as the mower was driven around. Then, outside of the active field, I was able to analyze the results of the data, work on my fused data, and play back the results of both the fused and unfused data in the visualizer. Also, the black box approach used in ROS allowed me to proceed with the localization and mapping system using the less accurate, unfused odometer data and then simply replace that service with the fused, more accurate odometer data when that service became available. No other components had to change, they just were able to immediately start using the more accurate odometry data.

2.3 Summary

This chapter provided an overview of the ASMO system. It described details of the hardware used in this project, including the mower itself, mechanization of the mower, and the addition of various sensors and microcontrollers for interacting with the hardware. The safety protocols implemented in the system were also outlined. A description of both the low-level microcontroller software as well as the high-level ROS system software was also provided. The following chapters dive deeper into the details of the core components implemented for the ASMO system.

Chapter 3

Throttle Control System

Throttle control of the engine is important because the amount of power needed to run the mower deck is much higher than the amount of power needed for moving the mower when the cutting blades are not engaged. Typical operation of the mower involves moving the machine from the garage to the yard using a minimal amount of power. Prior to starting the cutting blades on the mower deck, the operating manual recommends increasing the throttle to maximum. Computer control of the throttle allows the mower to move about the yard, navigating to a starting position, and then, when ready to start cutting, increase the throttle as the mower deck is turned on.

The throttle system detailed in this chapter involves both controlling the fuel flow to the gasoline-powered engine as well as monitoring the speed of the engine to ensure it maintains a consistent power output. In order to monitor the engine speed, a custom engine tachometer was built using magnets and a hall-effect sensor. Additionally, the original hardware governor was replaced with a software one so that the computer could maintain a constant and consistent power output from the engine. The software governor implemented for this project was based on a Proportional-Integral-Derivative (PID) loop.

3.1 Hardware

Control of the throttle system involves two key aspects. First, control of the throttle plate, which is what controls the flow of fuel to the engine. Second, monitoring the speed of the engine to ensure it maintains a constant operating speed. The original throttle control mechanism was a mechanical push-pull rod connected to a lever. The

speed of the engine was reported by a tachometer unit mounted on the control panel of the mower. Both of these items can be seen in Figure 55.

Figure 55. Original tachometer (1) and throttle (2)

The idea was simple: replace the existing push-pull cable with a servo and utilize the signal wire connected to the tachometer to provide an input to a microcontroller to count the pulses.

3.1.1 Throttle Plate Servo

As is the case with most riding mowers, the Bad Boy Outlaw XP mower comes with a throttle that is manually controlled using a lever attached to a push-pull cable. In

principle the idea was simple. Remove the push-pull cable and connect a servo to the throttle plate, as shown in Figure 56.

Figure 56. Throttle Control Servo

The servo used was a Hitec HS-5646WP, which has an IP67 rated waterproof case. IP stands for International Protection Code and the 6 means the case is dust tight while the 7 means the unit can withstand immersion in water up to 1 meter. I am not planning on operating the mower in the rain, but I thought having a dust-proof case would prove beneficial in the extremely dusty and dirty mowing environment. This servo has a stall torque of 157 oz/in at 6 VDC. The servo is controlled by a PWM analog output line on the Teensy 3.2 microcontroller located in the main computer unit. Servos are great little devices. You send them a pulse every 20 milliseconds that corresponds to the position you'd like the servo horn to be in and the servo drives itself to that position,

utilizing an internal potentiometer for feedback. Generally speaking, a pulse of 1ms wide represents 0 degrees, 1.5ms wide represents 90 degrees, and 2ms wide represents 180 degrees. According to the Hitec documentation, the HS-5646WP servo (Servo City, 2018a) used in this project responds to values from 750 microseconds to 2250 microseconds and has a stock travel of 118 degrees. The travel out of the box is 0.079 degrees per microsecond. The reported microseconds and degree per microsecond do add up to the reported travel:

2250 us – 750 us = 1500 us

1500 us * 0.079 degree/us = 118.5 degrees of travel

Once the servo was mounted in place, I connected a potentiometer to the Teensy 3.2 microcontroller to work as a manual throttle control. This was done to determine the upper and lower limits the servo needed to operate at. The idea was to have the PWM output track the input of the potentiometer, as read from one of the analog input pins. After some amount of empirical testing, I determined the servo's minimum throttle PWM value to be 1104 microseconds and the servo's maximum throttle PWM value to be 830 microseconds. Using the formula from before, the degrees of travel can be computed:

1104 us – 830 us = 274 us

274 us * 0.079 degrees/us = 21.646 degrees

This operating angle did correspond with what I visually observed as the throttle movement. While the servo is rated for 6 VDC operation, I am running the servo slightly underpowered at 5 VDC, but during my testing it had no problems moving the throttle plate. Running the servo at 5 VDC proved simpler than adding an additional 6 VDC power supply to the already cramped case.

The amount of force required to move the throttle plate is miniscule compared to the amount of force required to move the throttle control rod. The reason is that the control rod does not connect to the throttle plate directly, but instead is only indirectly connected through a hardware governor system built into the engine itself. The governor system is similar to a cruise control system in a car. It is designed to automatically adjust the throttle plate based on a set point—the set point in question being the location of the throttle control rod. The Bad Boy Outlaw XP mower used in this project has a Kawasaki 852cc engine, equipped with a pneumatic governor system. This means there is a fan mounted to the flywheel that spins with the engine. As the load on the engine decreases, the engine spins faster, causing the air blown by the flywheel to increase. This in turn causes the governor to pull the throttle plate towards the closed position, slowing the engine (Dempsey, 2007). As such, it is able to maintain a constant engine speed, as measured by Revolutions Per Minute (RPM).

Bypassing the original push-pull rod meant bypassing the original hardware governor system. I initially tried to attach the servo directly to the linkage the push-pull rod was connected to, but it required too much force to move. Connecting the servo to the throttle plate directly was the only option that didn't require utilizing the original push-pull rod. Bypassing the governor system in the engine meant I needed to accurately be able to read the engine speed in order to maintain a desired RPM under a variable load. The amount of fuel required to maintain a constant engine speed varies more than just when the cutting blades on the mower deck are enabled. It can also vary during turning operations and going up-hill. Additionally, the amount of power required by the engine to maintain a constant speed can even vary when the mower deck is enabled. For

instance, the power requirements increased when the mower encountered a thicker patch of grass.

3.1.2 Engine Tachometer

I expected the tie in to the existing tachometer to be straightforward. There was only one wire going to the small tachometer mounted on the control panel. Surely it was just picking up pulses from the engine that I could also easily read with a microcontroller. This proved more difficult than imagined. I traced the wire from the tachometer to the engine and it appeared to be connected to the engine's magneto. In order to verify how the signal on the tachometer wire worked, I connected it to an oscilloscope. The output from the magneto varied from -70 to +70 volts. That was problematic as the Teensy microcontroller could only handle 3.3 VDC signals. I used a diode to rectify the signal and clip off the negative voltage. This left me with a signal that travelled from 0 to +70 volts. That was better, but still not what I needed. In order to turn the signal into TTL level signal I could feed into the microcontroller, I ran the signal through an optoisolator that was rated for 200 V. An optoisolator isolates an input signal from an output signal using a light emitter and detector pair. This protects the output circuitry from the input circuitry, allowing for different ranges on the input and output. In my case the input was from 0 to +70 volts while the output was 0 to 3.3 VDC. This gave me what I thought was a clean signal. Next, I hooked the line up to a digital input on the Teensy and was able to read the inputs using an interrupt handler attached to the input pin.

Prior to running my tests, I recorded the signal widths from the magneto at various engine speeds for use as a baseline with the microcontroller readings. As I reran the tests with the microcontroller connected, I noticed I was getting correct readings

about 75% of the time when the engine was idling at 1800 RPM. As the engine RPM increased, the errors also increased. I spent a great deal of time troubleshooting this issue. At first, I was convinced there was a conflict between the interrupt I was using and either the serial output used for debugging or perhaps with the servo control. The servo is controlled through a PWM output that is managed by another interrupt in the system to keep the output pulses to the servo at specific levels. I meticulously went through and eliminated the various possibilities, even adding in a faster microcontroller—all to no avail. I could not remove the spurious readings. In order to further troubleshoot, I hooked the oscilloscope back up to the magneto wire to re-check the pulses. I noted something odd with the signal. As shown in Figure 57, there was a signal peak and then another small peak after the main peak.

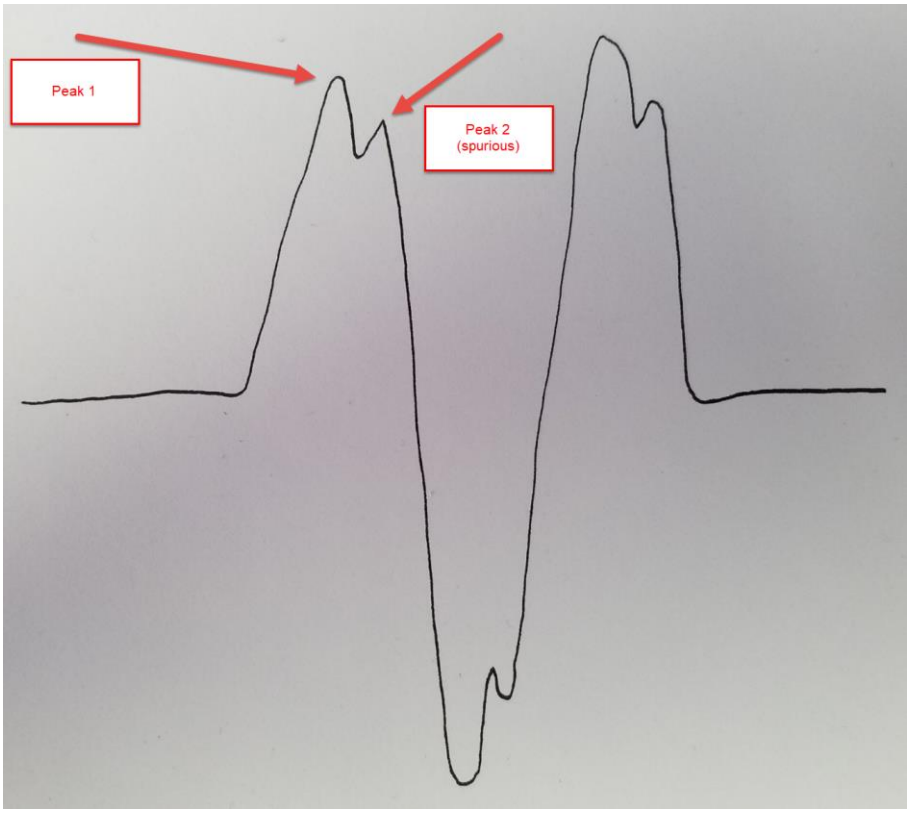


Figure 57. Engine magneto signal

My thought was that the second peak shortly after the main peak was causing the spurious readings and that I was only sometimes picking up this second peak. This would account for the very short readings I would sometimes capture. I did additional research on the control unit that came with the mower showing the tachometer reading. I wasn't able to find the exact details of how it worked, but I believe it is treating the signal as some sort of radio wave. The control unit is entirely plastic, powered by its own battery and it only has one wire going to it (the magneto wire). So, it has no ground reference. It also stopped working once I clamped the negative side of the signal with the diode. I gave consideration as to how I might be able to further clean up the signal.

After spending about a week on this issue, I decided the signal coming from the magneto was just not going to work. I had been working with Hall Effect sensors in the LIDAR and wheel encoder systems and decided to utilize one to determine the engine speed as well. A Hall Effect sensor can output a voltage corresponding to the strength of a magnetic field. The Hall Effect sensors used in this project are actually Hall Effect switches, meaning they turn on in the presence of a magnetic field and turn off when it is removed. My idea was similar to that of the magneto, except I would be adding magnets and the pickup sensor instead of utilizing the existing magneto system. This gave me more control over the output signal. The Hall Effect switch used in this project is the Melexis US5881 Unipolar Hall Effect Switch (Melexis, 2018) and can be seen in Figure 58.

Figure 58. Melexis US5881 Hall Effect Switch (Melexis, 2018)

I created a mount for the Hall Effect switch, which can be seen in Figure 59.

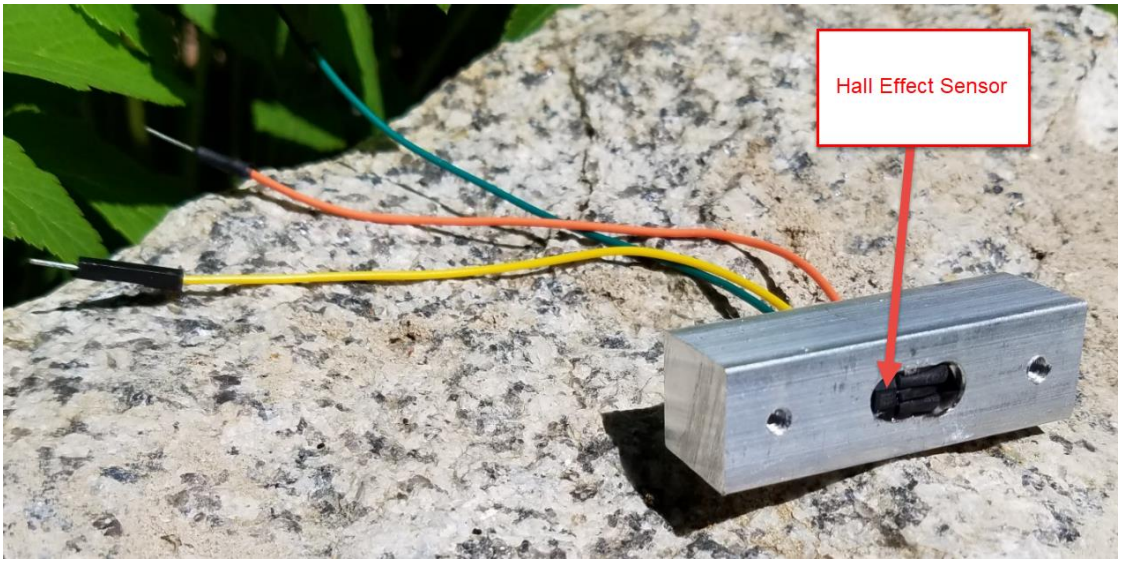


Figure 59. Hall Effect Sensor epoxied in a custom milled aluminum bracket

I mounted two magnets to the fan shroud of the engine, as shown in Figure 60.

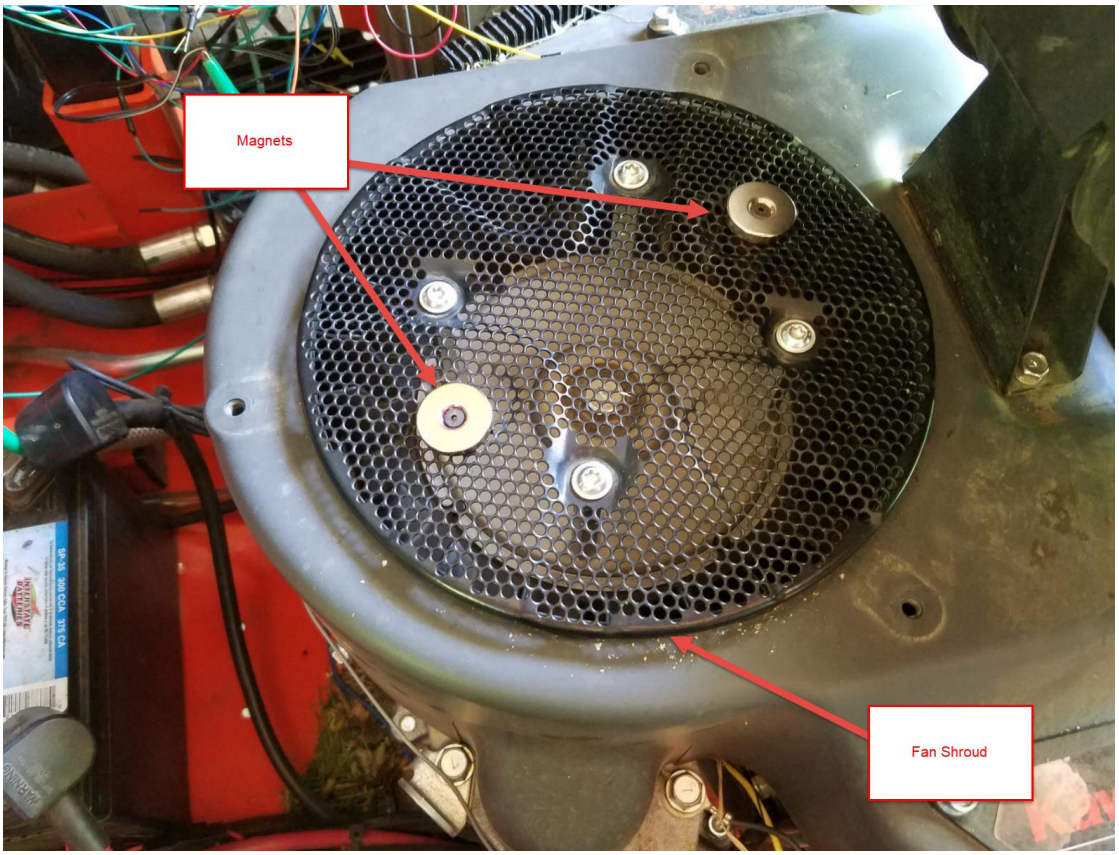


Figure 60. Fan shroud with magnets attached

While I could have used one magnet, I chose to use two to better balance the fan. I then attached the Hall Effect switch mount I made to the engine fan grill. Figure 61 shows the hall-effect sensor mounted to the fan grill above the shroud.

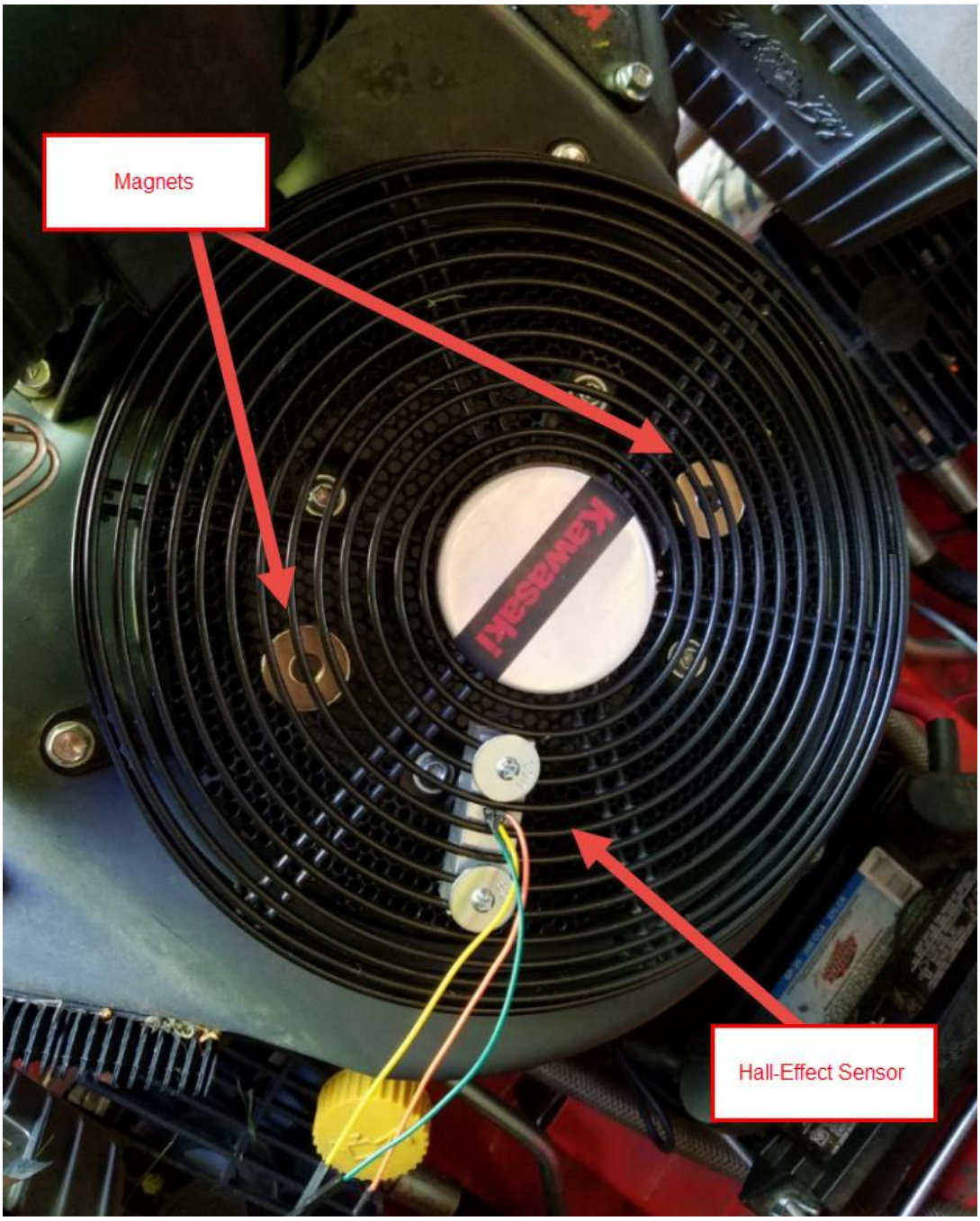


Figure 61. Engine Speed Sensor

This solution worked well and produced a clean output that I could read using the microcontroller without any glitches. The clean signal is shown in Figure 62 as seen on the oscilloscope. It should be noted that the signal shown is twice that of a single revolution of the engine crank shaft as there are two magnets present on the fan shroud. This means that I am receiving twice as many inputs as I would have previously received using the magneto.

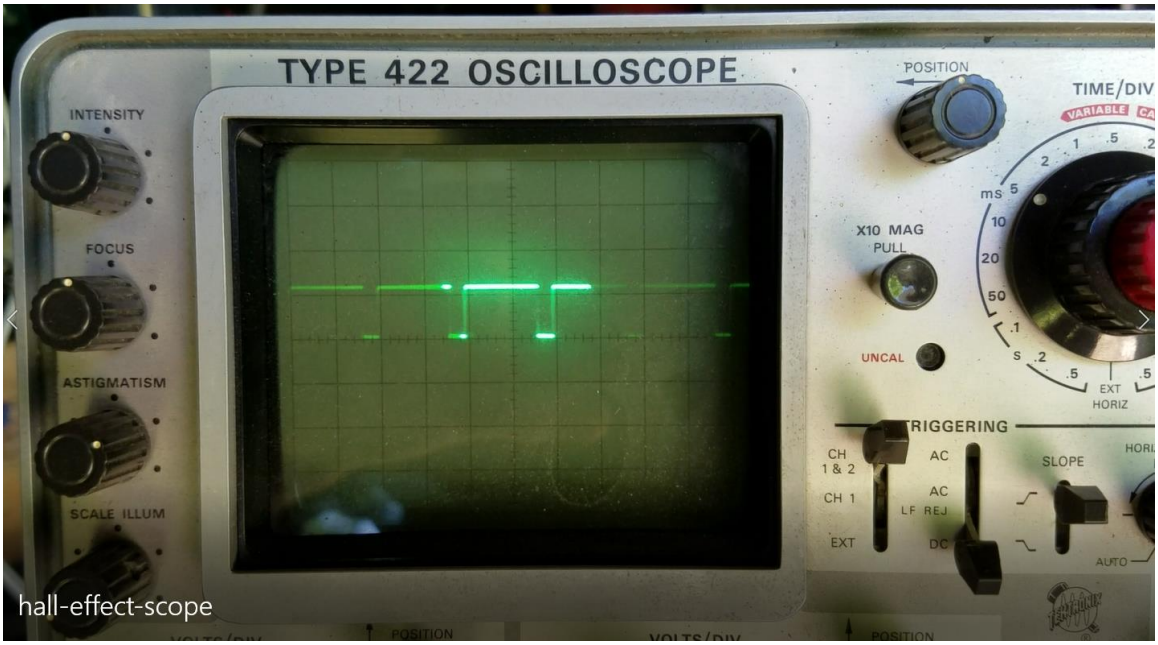


Figure 62. Hall Effect tachometer output

3.1.3 Microcontroller

Initially I planned to use a Adafruit Metro for the throttle control system. However, this was replaced with a faster and more robust Teensy 3.2 board during troubleshooting. Once I learned the Adafruit Metro boards did not have enough memory to run the ROS Serial library, I decided to keep the Teensy 3.2 board in place. This allowed me to expose the throttle directly to the ROS environment, without having to create an additional ROS service. The Teensy 3.2 runs faster than the Adafruit Metro

board, which means it can more easily implement the PID loop necessary for robust throttle control.

The Teensy 3.2 board contains a 3.3VDC ARM Cortex-M0 processor running at 72 MHz. All the digital input pins on this board can be used for interrupts. The Hall Effect sensor is a Melexis US5881 Unipolar Hall Effect Switch (Melexis, 2018). It has an operating voltage range from 3.5 VDC to 24 VDC. This sensor requires more voltage than the Teensy board, but another advantage of the Teensy 3.2 board is that the digital input lines are 5 VDC tolerant. This means I was able to run the Hall Effect sensor at 5 VDC and feed the output directly into the Teensy board. This particular Hall Effect sensor works as a switch, with a typical operating point of 25 mT (Milli-Teslas) and a release point of 20 mT. The magnets used were Grade N42 with a rating of 1320 mT. This allowed the sensor to be mounted approximate ¼" above the magnets while still picking up the magnetic field reliably.

3.2 Software

The software for the throttle control system was implemented on a Teensy 3.2 microcontroller. The Hall Effect switch was read using an interrupt handler while the main program loop utilized a Proportional-Integral-Derivative (PID) algorithm to implement the software governor to maintain a consistent power output. Lastly, the throttle control system was exposed as a ROS service to the rest of the ROS environment, so the throttle level could be adjusted dynamically.

3.2.1 Hall Effect Switch Interrupt Handler

The Hall Effect switch is connected to a digital input pin on the Teensy 3.2 board. The digital input is set up to have a software interrupt handler run on the falling edge of the incoming pulse on the input pin. This occurs whenever the magnet passes under the Hall Effect sensor. The Melexis Hall Effect sensor has a maximum switching frequency of 10 KHz (10,000 times per second), which is well above the engine operating speed. The engine RPM was recorded using the original tachometer that came with the mower. The engine was noted to idle at 1,800 RPM. When the original throttle lever was moved to its full position, the engine's noted speed was 3,600 RPM. These values became the lower and upper limits used in the software throttle control system. A maximum RPM of 3,600 means the maximum number of switches per second for the Hall Effect switch can be calculated as follows:

3,600 Revolutions/Minute ÷ 60 Seconds/Minute = 60 Revolutions/Second

Since there are two magnets mounted on the fan shroud, each revolution is going to generate two pulses from the Hall Effect switch. A switching frequency of 120 times per second is well below the 10,000 times per second rating of the Hall Effect switch.

An individual pulse on the input pin is not of much use on its own. What is useful is the amount of time between pulses, which can then be turned into an RPM value. To that end, the job of the interrupt service handler is simply to record the time at which the interrupt occurred. This is in accordance with the idea that interrupt service routines should be as short as possible so as not to interfere with the normal operation of the system. The Arduino programming environment has a function that is frequently used for timing operations called millis. According to the Arduino documentation, the millis

function “returns the number of milliseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.” There is also a micros function that returns the number of microseconds since the Arduino board began running the current program. The micros function rolls over after approximately 70 minutes. Both millis and micros return unsigned long data types.

In order to compute the time between pulses inside the interrupt handler, we simply need to subtract the current pulse time from the last pulse time, and then update the last pulse time with the current pulse time. In order to know which function to use, millis or micros, some calculations should first be done to determine what the expected value ranges are for the various engine speeds. As previously stated, when the engine is spinning at 3,600 RPM it is spinning 60 times per second and generating 120 pulses per second. This can be translated into the number of milliseconds between pulses as follows:

1,000 milliseconds/second ÷ 120 pulses/second = 8.33 milliseconds/pulse

So, at the engine’s top speed, there will be 8.33 milliseconds between pulses. While 3,600 RPM represents the engines recommended top-speed, it is not the actual top-speed of the engine. In order to be robust, the software should be designed to handle values larger than those expected to be seen. Doubling the 3,600 RPM value to 7,200 RPM means there will be 240 pulses every 1 second, which means there will be a pulse every 4.17 milliseconds. While technically the millis function should suffice for measuring this amount of time between pulses, it is very close to its 1 millisecond resolution. Additionally, microcontrollers do not handle floating point numbers well. 240 pulses every 1 second can be more accurately represented without floating point numbers by

using microseconds. 240 pulses every 1 second means there will be 1 pulse every 4,166 microseconds. For these reasons, I chose to use the micros function instead of millis.

One advantage of the millis function is that it overflows every 50 days. Since the mower would never be operating continuously for this long, I would not have had to manage an overflow value when computing time between pulses. However, the micros function overflows approximately every 70 minutes, which is certainly within the operational time of an afternoon's worth of mowing activity. Therefore, it is important to handle the possibility of an overflow. This is done by simply filtering out invalid values. This is not done in the interrupt handler itself, which has been kept deliberately simple. The interrupt handler simply records the result of the micros function in a variable called tachReading and then exits. It does not matter when this reading is read—the idea is that the most recent reading is always available in the tachReading variable. There is no attempt made to handle invalid values, due to overflow or errant signal reads. These various possibilities are handled in a separate function, called updateCurrentTach, that runs periodically, but at a much slower rate.

The reason for this decoupling is that interrupt service routines should be kept as short as possible so as not to interfere with the rest of the system. There are additional functions that need to occur when updating the currentTach variable used by the rest of the system. For instance, transitioning the state of the engine from STOPPED to RUNNING or from RUNNING to STOPPED generates events that need to be sent to the rest of the system over the serial port. Communicating over the serial port is inherently slow and should not be done inside the interrupt service handler. The updateCurrentTach

function handles validating the tachReading variable, updating the currentTach with tachReading, if tachReading is valid, as well as transitioning the engine state.

3.2.2 Servo Control Algorithm

Once the engine speed was reliably obtained, the next step was to dynamically adjust the throttle plate based on the current engine load. Setting the throttle plate to a particular position will result in a particular engine speed. However, as the mower encounters additional loads such as hills, thick grass, turning, etc., the engine speed will decrease. In these circumstances, the throttle plate will need to be opened further, providing more fuel to the engine, thereby driving the engine to restore its previous speed. Once the desired speed has been reached, the throttle plate should maintain its position until the load changes again. As previously mentioned, this is the job of the engine governor and is based on a feedback loop.

Given what seemed like a fairly simple task, my first approach involved simple "if" statements to keep the engine speed at a particular value. Meaning, *if* the actual speed dropped below the desired speed, open the throttle. Likewise, *if* the actual engine speed rose above the desired engine speed, close the throttle. While this worked, it was not robust enough to handle the dynamically changing environment without having to add new checks every time a previously unconsidered situation arose, which was a frequent occurrence. The simple "if" statement approach also resulted in a lot of cycling where the engine would continually overshoot or undershoot the target value. This cycling could be accounted for, but it led to a brittle solution because it continually needed adjusting. I knew there had to be a better approach.

The type of feedback control system used to govern engine speed is an ideal candidate for a Proportional-Integral-Derivative (PID) controller. This type of controller is used in diverse systems such as automotive cruise control (Pradhan, 2017), and furnace temperature control (Guofang, 2012). The basic overview of a feedback system utilizing a PID controller can be seen in Figure 63.

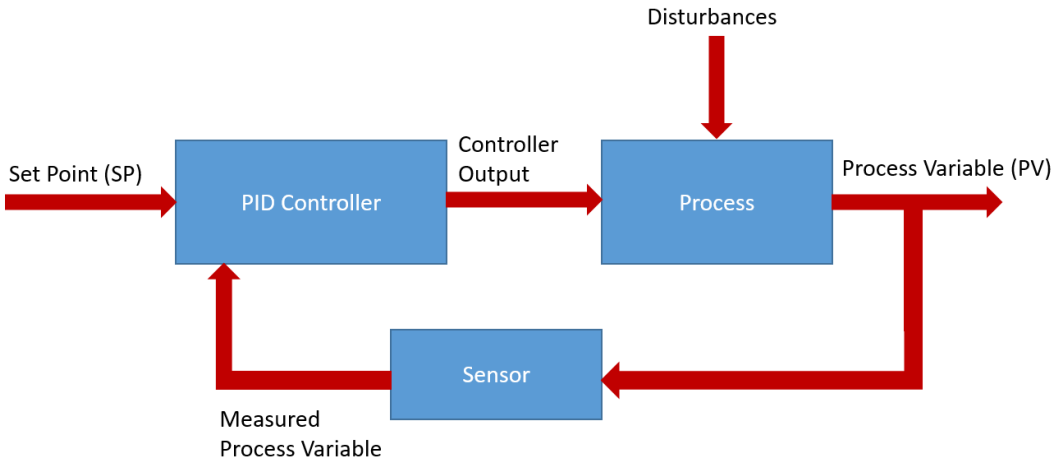


Figure 63. Typical PID control loop

The basic idea of the process under PID control is that the PID controller looks at the Set Point and compares it with the Measured Process Variable value. If they match, the system does nothing. If they do not match, the PID controller adjusts the output and then rechecks the Measured Process Variable value. The system continually adjusts the output in a positive or negative fashion in an attempt to drive the Measured Process Variable as close as possible to the Set Point. How quickly the output changes and by how much are parameters that need to be configured in the PID controller. This process is represented mathematically in Figure 64.

$$\text{Output} = K_P e(t) + K_I \int e(t)\,dt + K_D \frac{d}{dt} e(t)$$

Where : e = Setpoint - Input

Figure 64. The standard PID equation (Beauregard, 2011)

The Proportional part of the PID algorithm means that the Controller Output should be adjusted in proportion to the amount of error in the system. The error is defined by subtracting the Measured Process Variable (Input) from the Set Point. Basically, it's the difference between the desired engine speed and the measured engine speed. This error value is designated as *e* in the PID equation. The Proportional part of the PID equation is the first part: Output = $K_P e$. The term K_P is a constant that represents the Proportional coefficient, which is an unchanging value determined during the initial tuning of the PID controller.

The second part of the PID algorithm is the Integral. The Integral is used to accumulate the amount of error over time. The accumulated error is then multiplied by an Integral coefficient, K_I. The Integral coefficient is another constant whose value is determined during the initial tuning of the PID controller. The Integral part of the PID equation is the middle part: $K_I \int e dt$. The Integral portion of the PID controller allows the system to overcome steady-state error. Steady-state error occurs when the error is very small in relation to the desired output. This state results in the error itself not having much of an effect on the output, making it steady. The *e*dt part of the equation accumulates these small errors over time, allowing them to have an effect on the output. How much of an effect is determined by the Integral coefficient, which gets multiplied with the accumulated errors. The resulting value then becomes large enough to have an effect on the output.

The last part of the PID algorithm is the Derivative. The Derivative component is not always used when implementing PID controllers, but it can be useful when there are large changes that can be periodically introduced into the process. For instance, if the process under control is a burner on a stove keeping water boiling, this process could be controlled using a PI controller. However, if chunks of ice are periodically introduced into the boiling water, it would result in a large temperature change with an associated large error (Schaenzle, 2016). In a normal PI controlled system, it would take a while for the output to be affected, since the error is accumulated and compensated for over time. The Derivative portion of the equation allows the system to quickly account for sudden changes in the system. The Derivative part of the PID equation is the last part: $K_D(d/dt)e$.

3.2.3 Servo Control PID Implementation

There are three steps to implementing any PID controller. The first is mapping the PID algorithm to the process in question. The second is writing the code and the third is tuning the controller to find the correct values for the coefficient variables K_P, K_I, and K_D. The coefficient variables are similar to the springs in the hardware governor system installed on the engine itself. Figure 65 shows how the throttle control system in the ASMO project has been mapped to the PID algorithm.

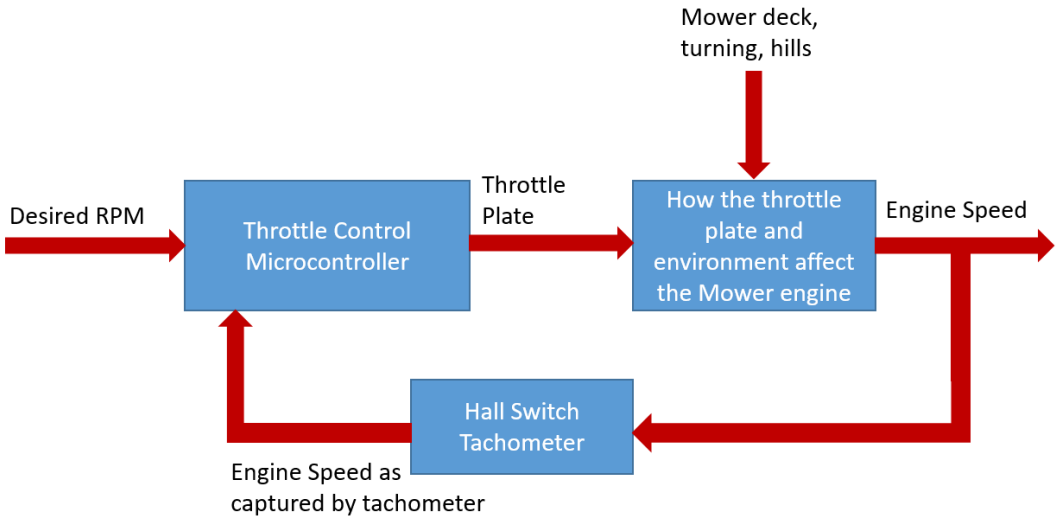


Figure 65. PID control loop as implemented

The Desired engine speed (RPM) is the Set Point. The throttle plate, as controlled by the throttle servo, represents the Controller Output. The disturbances come from any environmental condition that causes additional load on the engine. The additional load could be caused by the mower suddenly encountering a thicker patch of grass, from having to travel up an incline, or even simply by making a turn. The process itself is how the throttle plate and the environment affect the engine. The sensor in this case is the represented by the Hall Switch and magnets that were added to the engine fan shroud and grill. The PID Controller is implemented by the microcontroller acting as the Throttle Control System.

Implementing the Derivative aspect of the PID algorithm can be done in a fairly straightforward manner. It requires determining the amount of error, which is simply the difference between the desired engine speed (setpoint) and the measured engine speed (currentTach). Then, this error is multiplied by the Proportional coefficient (Kp). An

implementation of the Proportional aspect of the PID controller can be done using the following pseudo-code:

```
error = setpoint - currentTach
output = error * Kp
```

Next, in order to implement the Integral aspect of the algorithm, the error needs to be accumulated over time (accumulatedError). The amount of time the error needs to be accumulated over is the time since the last error accumulation to the current error accumulation (deltaTime). The following pseudo code shows how the error can be accumulated and added into the output:

```
accumulatedError += error * deltaTime
output = (error * Kp) + (accumulatedError * Ki)
```

The last part of the PID algorithm is the derivative. The derivative represents the slope of the curve, whose points derive from a plot of the error over time (derivativeError). This value is then multiplied by the Derivative coefficient. This basic PID implementation can be expressed using the following pseudo code.

```
accumulatedError += error * deltaTime
derivativeError = (error - lastError) / deltaTime
lastError = error
output = (error * Kp) + (accumulatedError * Ki) + (derivativeError * Kd)
```

I did testing with the above implementation, but there are additional aspects needing consideration that are not taken into account.

One problem I ran into has to do with a phenomenon known as Derivative Kick. As mentioned, the Derivative part of the PID is meant to account for sudden changes to the steady state. However, when a large, sudden change occurs, it results in a very large change in the output as the controller attempts to compensate. This large change in output is known as Derivative Kick (Beauregard, 2011). It can be accounted for by

modifying the equation, but this is not the only issue with the basic PID implementation. There are other aspects to be considered as well. For instance, when doing on-the-fly tuning, there is the need to smooth drastic output changes that can result from modifying the coefficient values.

Additionally, avoiding a phenomenon known as Reset Windup is another issue. Reset Windup occurs when the PID controller thinks it can do something that it cannot—such as output a value greater than the PWM output can handle. While clamping the output to a certain value will prevent an invalid value from being used on the PWM line, the PID controller would continue to try and increase the output, until eventually it had an effect. Once the output does manage to affect a change, it will take a while for the PID controller to recover from the higher value it was trying to output. This behavior is known as Reset Windup. It too can be accounted for, but when doing the research into these various issues, I discovered that the PID library for the Arduino takes these issues into account, making a nice set of functions available that I could expose to the rest of the ROS environment through a ROS service. The PID library for the Arduino environment was implemented by Brett Beauregard (Beauregard, 2011).

As can be seen in the PID equation and pseudo code, choosing the right coefficient values is very important to the behavior of the PID controller. The process of finding the right values is known as tuning. There is no set formula to tune a PID controller since each PID controller is highly specific to the environment in which it operates. That said, looking at how each coefficient is mapped to the process in the ASMO project helped guide me in choosing appropriate starting values. The Proportional coefficient represents how much of a change the PID controller should make

when adjusting the output—should the throttle plate be moved a short or a long distance? In my testing, small changes to the throttle plate represented relatively large changes in the engine speed, so I knew I wanted this number to be small. The Integral coefficient represents how quickly the PID controller should respond to changes—how long should the PID controller give the engine to adjust its output based on a throttle plate change? Based on my testing, the engine's response to a throttle plate change appeared nearly instantaneous. For instance, I could manually adjust the throttle plate to get the engine to "rev" up and down. However, what appeared immediate to me was not immediate in terms of microcontroller timing. Based on my testing, sampling the errors every 100 milliseconds and allowing the Integral value to be relatively small, and worked quite well.

The Derivative coefficient represents how much of an effect on the output a sudden load change should have. Not all changes in load were large enough to benefit from the Derivative coefficient. In my testing, performing a zero-turn did not generate sufficient load on the system. However, switching the mower deck on did cause a large sudden change in load, and in this case the Derivative coefficient actually had some bearing. While the Derivative coefficient did have some value for certain operations, it was not required all the time. That being the case, to keep the tuning process simple, I started adjusting the values of just the Proportional and the Integral coefficients, leaving the Derivative coefficient alone as that could be added in later. From there, it was a matter of adjusting the values until the system operated well—meaning, it held close to the desired engine speed under a variety of loads with limited cycling. Once the system

operated similarly to how it operated with the hardware governor, I stopped adjusting the values.

3.2.4 Custom ROS Throttle Service

The strength of the ROS environment was evident when developing the Throttle Control system. The componentized nature of ROS allowed me to focus on encapsulating a software component for managing the throttle system. I was able to expose useful parts of the system publicly as ROS Topics for consumption from other services, including command line tools. These Topics are real-time messages sent by the throttle system, which allowed me to monitor the running state of the service from a terminal session. Table 5 contains the list of ROS Topics created for the Throttle Control system. The Topic Frequency represents how often the message is published to the ROS Topic.

Table 5. Throttle Control ROS Topics

Topic Name	Topic Type	Topic Frequency	Topic Description
/throttle/tach	Int16	1 second	The speed of the engine as represented by revolutions per minute.
/throttle/hall	Int32	200 milliseconds	The average number of microseconds between pulses of the Hall Switch sensor.
/throttle/engine	Int8	On Change	An enum representing the current state of the engine: 0 – Stopped 1 – Cranking 2 – Running

I was also able to expose a set of ROS Services, which allowed me to interact with the throttle system from a terminal window as well as from other ROS components. A ROS Service represents a function call. Since these functions are publicly exposed to the ROS environment, they can be invoked from a terminal window. This makes them useful not just for programmatic interaction with other components in the system but also for debugging purposes. For instance, I could increase the engine throttle position by pressing the Throttle Up button on the User Control Panel (see chapter 6) or by invoking the setThrottle Service from a terminal window. The list of Services implemented for the Throttle Control system is outlined in Table 6.

Table 6. Throttle Control ROS Services

Service Name	Service Description
setThrottle	Takes a single Int8 value as a parameter representing the new throttle position. Valid values are 1 through 10 and map to the following engine RPMs: 1 – 1600 2 – 1800 (Idle) 3 – 2000 4 – 2200 5 – 2400 6 – 2800 7 – 3000 8 – 3200 9 – 3400 10 – 3600
getThrottle	Returns an Int8 value representing the engine speed.
getEngine	Returns an Int8 value representing the current state of the engine. Valid values are: 0 – Stopped 1 – Cranking 2 – Running

Additionally, ROS is dynamic in that it has a parameter system, which allows a programmer to expose an externally configurable set of values to the rest of the system. The parameter system includes interactive tools that allow these configurable values to be manipulated at runtime from a terminal session. This worked very well in the case of the throttle control system as the PID coefficients could be interactively adjusted from a terminal window. I could then obtain immediate feedback from the engine by listening to it but also by monitoring the engine speed from another terminal window that was outputting values from the /throttle/tach Topic. This allowed me to quickly adjust the coefficients in real-time. The list of ROS parameters for the Throttle Control service are listed in Table 7.

Table 7. Throttle Control ROS Parameters

Parameter Name	Parameter Type	Parameter Description
/throttle/pid_Kp	Int32	The Proportional coefficient value.
/throttle/pid_Ki	Int32	The Integral coefficient value.
/throttle/pid_Kd	Int32	The Derivative coefficient value.
/throttle/pid_setpoint	Int32	The desired engine speed.
/throttle/pid_sampletime	Int32	The number of milliseconds between sensor updates to the PID, which also represents how often the output may change.
/throttle/pid_min	Int32	The minimum output value of the PID controller.
/throttle/pid_max	Int32	The maximum output value of the PID controller.
/throttle/pid_automatic	Bool	Indicates whether or not the PID controller should be running. If true it is updating its internal state and adjusting the output.

Note that the ROS Parameter Server utilizes the XML-RPC protocol and is therefore limited to the data types supported by XML-RPC. This is the reason for the use of the Int32 data type rather than the Int16 and Int8 types used in the throttle Topics.

Chapter 4

Laser Distance System

ASMO requires knowledge of the environment and its immediate surroundings in order to perform autonomous mowing operations. The robot's primary means of obtaining environmental knowledge is through a laser distancing system. This distancing system is capable of providing 360-degree centimeter-level distance information between the robot base and objects in the environment, up to 40 meters away. This distancing information is gathered several times per second and is used to build maps, plan navigation, as well as avoid objects in the robot's path.

The laser distance system built for the ASMO project is unique in that it is a low-cost alternative to more expensive laser distance systems used in most academic projects, which can retail for over $5,000. The individual LIDAR (Light Detection and Ranging) units outlined in this chapter can be built for under $200 in parts. Three individual LIDAR units are used in the ASMO project, providing 360-degree distance information around the entire perimeter of the mower. This is significantly better than the 90-180-degree coverage offered by the more expensive scanning units (Mertz, 2013). The downside to this custom-built laser system is that it operates at a 2 Hz instead of the 40 Hz seen by the more expensive units. However, as will be discussed later in this chapter, there are techniques that can be implemented to help speed up the sensor for certain operations, such as obstacle detection. Additionally, as will be shown in subsequent chapters, the slower speed of the custom laser system does not affect its usefulness for navigation operations.

4.1 Hardware

The LIDAR unit built for this project is composed of an Adafruit Metro microcontroller (Adafruit, 2018a), an Adafruit TB6612 DC/Stepper Motor driver breakout board (Adafruit, 2018e), a Garmin Lidar Lite v3 laser sensor module (Garmin, 2018a), and a NEMA 14 Bipolar stepper motor (Amazon, 2018b) used to rotate the laser 360 degrees. There were a handful of mechanical components required such as a timing belt, timing pulley, and bearings. Additionally, raw aluminum stock was machined in order to create the mechanical components necessary to house and drive the sensor.

The heart of the LIDAR unit is the laser sensor itself. The laser sensor used in this project is the Lidar Lite v3 optical distant measurement sensor from Garmin (Figure 66). At the time of this writing, the sensor sells for $129.99 USD on the Garmin web store (Garmin, 2018a).

Figure 66. Lidar Lite v3 Sensor Module

The sensor operates at 5 VDC (135 mA) and communicates with a microcontroller over I2C. The sensor is capable of measuring distance accurately up to 40 meters at 270 times per second. The sensor measures the distance between itself and a single target. The sensor's performance characteristics can be seen in Table 8.

Table 8. Lidar Lite v3 specifications (Garmin, 2018b)

Specification	Measurement
Range (70% reflective target)	40 meters (131 feet)
Resolution	+/- 1 centimeters (0.4 inches)
Accuracy < 5 meters	+/- 2.5 centimeters (1 inch) typical Non-linearity present below 1 meter (39.4 inches)
Accuracy >= 5 meters	+/- 10 centimeters (3.9 inches) Mean +/- 1% of distance maximum Ripple +/- 1% of distance maximum
Update rate (70% reflective target)	270 Hz typical 650 Hz fast mode (reduced sensitivity) > 1000 Hz short range only
Repetition rate	~50 Hz 500 Hz max
Wavelength	905 nm (nominal)
Total laser power (peak)	1.3 W

To better understand how this sensor operates, it is useful to visualize the accuracy as a beam. The beam is fairly uniform with a 1" width for the first 39.4". After the first 39.4", the beam widens on average ~1% of the distance. This means at 40", the beam width would be 0.4" and at its maximum sensing distance of 1,572" (131') the beam would be 15.72". Page 12 of the operating manual states this another way, "At very close distances (less than 1 m) the beam diameter is about the size of the aperture (lens). For distances greater than 1 m, you can estimate the beam diameter using this equation: Distance/100 = beam diameter at that distance (in whatever units you measured

the distance).” Using this equation, we can take 131’ divide it by 100 to get 1.31’ or 1,572”, which is the same value obtained using the 1% of distance from the table.

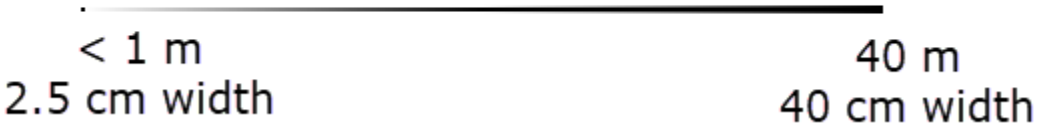


Figure 67. Laser beam width to distance relationship

This laser-based distance sensor is fast and works well outdoors under various lighting conditions. I performed a variety of tests on both long and short-range targets outdoors in various lighting conditions that would be representative of mowing conditions. For instance, I tested 10’, 50’, and 100’ distance to a tree, a building, and a rock wall under sunny, cloudy, and partly cloudy conditions. During each test, the average distance reported over 1,000 scans were within the margin of error stated for the sensor.

While the sensor itself performs well, the narrow beam does not provide sufficient detail of the surrounding environment to produce a map or to perform navigation using the ROS navigation stack. In order to accumulate additional detail, I decided to take the approach used by the more expensive sensors and perform a 360 degree sweep of the area. This approach would allow me to get a very detailed view of the area surrounding the mower. The more expensive LIDAR units keep the laser in place and use a rotating mirror to direct the laser beam in and out of the environment. I did not take this approach as I had concerns over whether or not I would be able to precisely align the mirror, keep it in place and clean during bumpy and dusty mowing operations. The more expensive LIDARs had IP67 enclosures to protect them from dirt and debris. These enclosures

were certainly part of the reason for the more expensive nature of these scanners. Rather than use a spinning mirror, I decided to spin the laser module itself.

There is a precedent for spinning a laser without the use of a mirror. The Slamtec RPLIDAR A1M8 (Slamtec, 2018) uses a belt drive to spin a laser module. It uses an optical encoder to count pulses to know where in the rotation the sensor is located. However, I was concerned about using a small pulley drive like the one used in the Slamtec device due to the vibrations encountered in mowing environments. If the belt slipped, it would throw off the timing in the system making it difficult to deduce where the sensor was in its rotation.

Figure 68. Slamtec A1M8

Instead, I decided to use a stepper motor and a timing belt to precisely rotate the sensor through a 360-degree sweep. This approach would yield a beam pattern such as that shown in Figure 69. Stepper motors are motors that can be precisely controlled through their entire rotation. Each stepper motor is rated for a certain number of steps that will move it through an entire rotation. I chose to use a 200-step stepper motor,

which means each step will move the motor 1.8 degrees (360 degrees / 200 steps). 100 steps will move the motor shaft 180 degrees. Using a stepper motor and robust timing belt would allow me to positively control the motor shaft, and thereby the laser sensor, without having to consider slip—timing belts have teeth specifically designed to prevent slip. I added a Hall Effect sensor and a magnet in order to know where a "home" position was for the laser. The home position represents where 0 degrees is located. Once home was known, I would know in which direction the laser was pointing so with each spin of the laser I could align distance readings by degree.

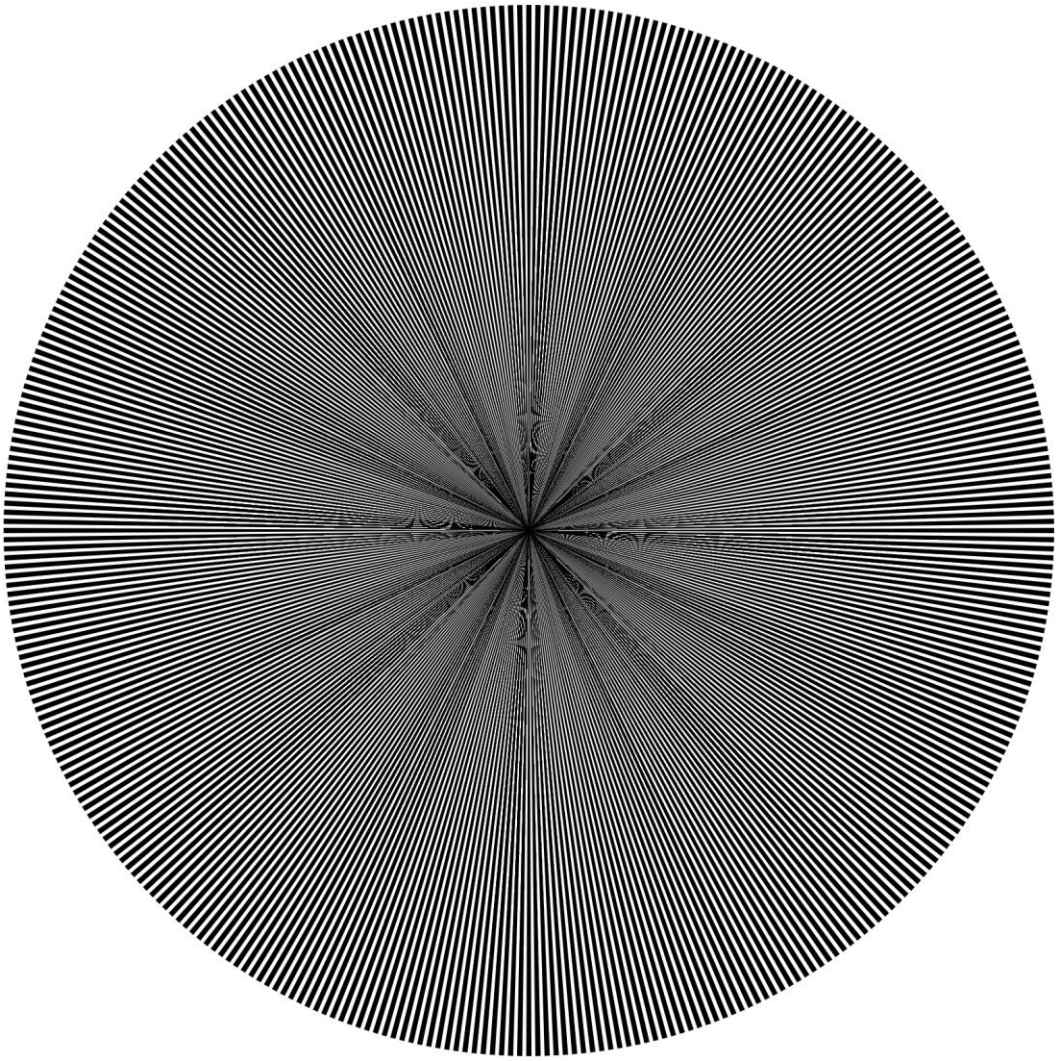


Figure 69. 360-degree sweep of Lidar Lite v3 beam (0-40 m)

The next item considered was where to mount the LIDAR unit. In order to get a 360-degree view surrounding the mower the LIDAR unit would need to be mounted high enough to not be occluded by the mower itself. Even with the roll bar down, the mower is still 4' tall. Mounting the LIDAR unit at this height meant it would miss many opportunities to see landmarks, especially the stone walls on my property which are generally less than 3' tall. Additionally, as the LIDAR was meant to be the primary means of both navigation and obstacle avoidance, mounting the LIDAR 4' off the ground was a safety concern as there are many dynamic obstacles under 4' tall such as children and pets. Due to these concerns, mounting the LIDAR lower to the ground is a requirement.

Mounting the LIDAR on the front of the mower would give me 180-degree view of what's directly ahead, but not provide relevant side views useful for mapping and localization. For these reasons, I decided to build three LIDAR units and mount them in a triangular pattern around the mower (Figure 70). This meant mounting two forward-facing LIDAR units on the front—one on the left side and the other on the right. A third LIDAR unit was mounted on the rear of the mower. This mounting pattern enabled a 360-degree view surrounding the mower that would be only 18" off the ground.

An advantage to the use of multiple LIDAR units is the cross-coverage. While these LIDARs operate at a slower frequency than more expensive units, the positive control over their spin allows them to be precisely interleaved. For instance, as the front-left LIDAR is completing its forward-facing sweep, the front-right LIDAR can begin its own forward-facing sweep of the same area. This approach effectively enables doubling

the scan rate of the forward-facing area. All three LIDAR scans are combined together in software to produce one virtual LIDAR scan. This process is discussed in section 4.2

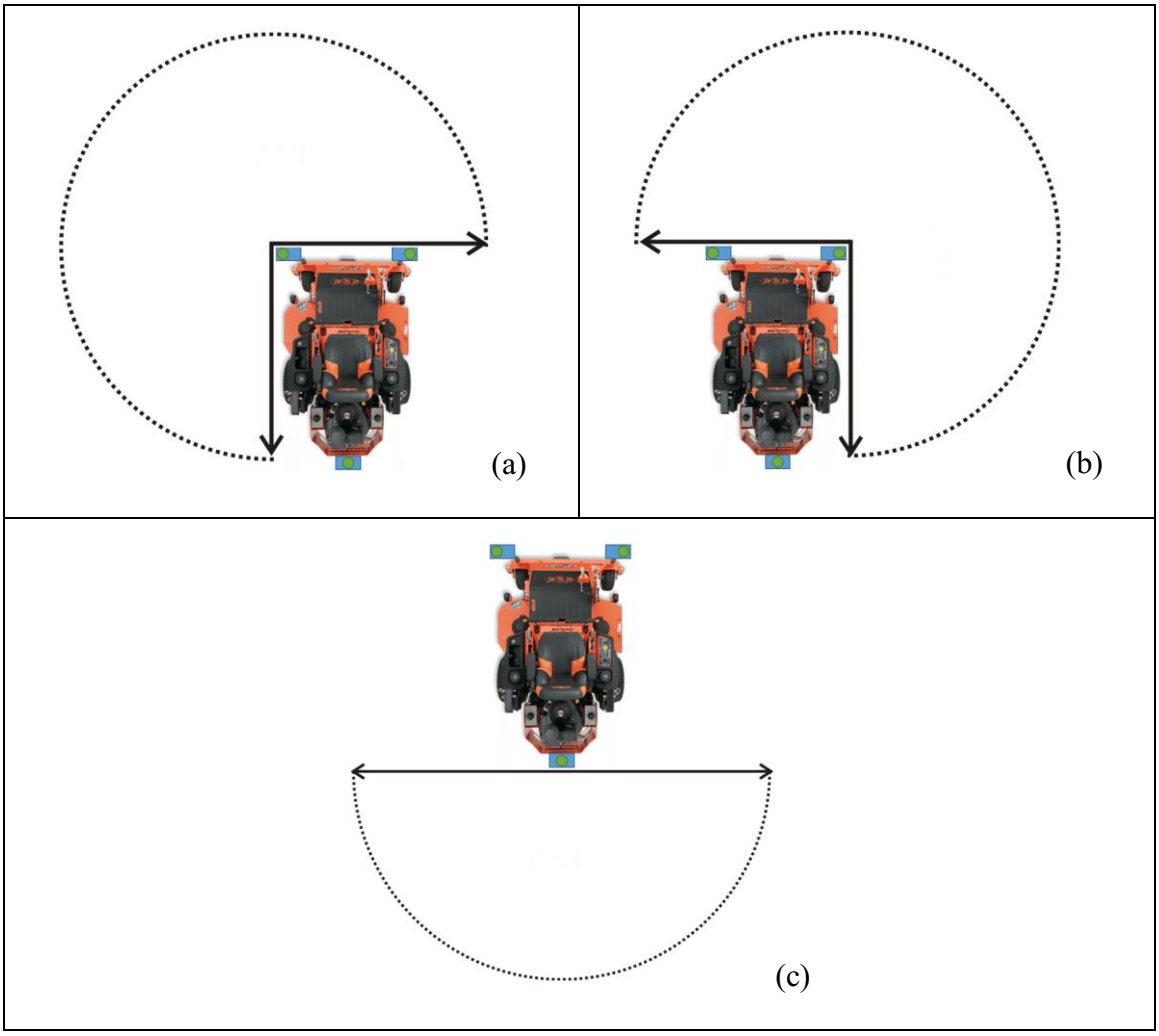


Figure 70. LIDAR unit coverage pattern. (a) front-left, (b) front-right, (c) rear

4.1.1 Mechanical Design and Build

Now that I knew the approach I wanted to take, the next step was coming up with a working design and building a prototype. Being able to directly control the motor shaft through its entire rotation is good, but 200 steps per resolution resulted in a rather broad 1.8 degrees per step. I decided to introduce a reduction mechanism to reduce each step to 1 degree. Taking this approach would allow 1 step to equal 1 degree and thereby 360

steps would directly correspond to one full 360-degree rotation of the laser sensor. This approach meant no floating-point calculations were necessary to determine the angular position of the laser sensor as it was rotated by the stepper motor—one step equaled one degree of movement, 12 steps equaled 12 degrees of movement, etc. Additionally, this approached provided more detail for each full sweep as each step moved the laser 1 degree instead of 1.8 degrees. In order to accomplish this reduction, I utilized a 10-tooth timing pulley on the stepper motor shaft, coupled to an 18-tooth timing pulley on the sensor shaft.

One of the main issues with rotating the sensor is the need for an electrical connection between the sensor, which is rotating, and the rest of the system, which is not rotating. The advantage to using a mirror would have been no electrical connections between the spinning head and the base unit. The laser sensor requires two wires for the ground and +5 VDC as well as two wires for the I2C connection. There are two additional wires on the laser itself, but they are not used in the I2C connection. After doing some research, I discovered an electromechanical device known as a Slip Ring. Adafruit happened to sell miniature 12mm diameter 6-wire slip-rings for $17.50 (Adafruit, 2018f). This slip ring can be seen in Figure 71.

Figure 71. 12mm diameter 6-wire slip ring

These slip rings allowed the top set of wires to spin while the bottom set remained stationary. This type of physical connection does represent a stress point in the system. I was not able to find any information regarding the lifespan of this particular slip ring. However, I was able to find information on similar slip rings manufactured with the same gold-plated brushes that had a rated life cycle of up to100 million revolutions (Deublin, 2018). In addition, the slip rings are rated for 2 Amps and 300 RPM, which is much higher than the required 0.135 Amps and 120 RPM of my system. The lower speed and power draw in my application should extend the life of the slip ring.

Once the general mechanical process had been determined, the next step was to create a prototype to verify that the basic idea would work. This prototype was more of a proof-of-concept than something I expected to mount on the mower itself. It was built with the idea to ensure the process I had envisioned would actually work and to determine the overall dimensions of all the various components, including wiring, so the final case could be constructed. The prototype also allowed me to experiment with the software needed to read the laser data and test encoding techniques.

One of the more difficult pieces of the prototype build process was creating the spindle that contained the slip ring and to which the 18-tooth pulley mounted. This was also the spindle to which the laser sensor would be mounted. This spindle had to be kept perfectly straight through a top and bottom bearing in order for the laser to spin parallel to the base plate. The only 18 tooth timing pulley I could locate had a maximum bore of 1/2". However, the outer diameter of the slip ring was 12 mm (0.472"), which meant the outside diameter of the spindle containing the slip ring was going to be larger than the top part to which the pulley was mounted. I ended up making the top part of the spindle 1/2"

and the bottom part of the spindle 5/8". This enabled me to use a standard 1/2" bearing to hold the top part of the spindle and a standard 5/8" bearing to hold the bottom. I was new to machining, so it took a significant amount of trial and error to create the spindle shown in Figure 72.

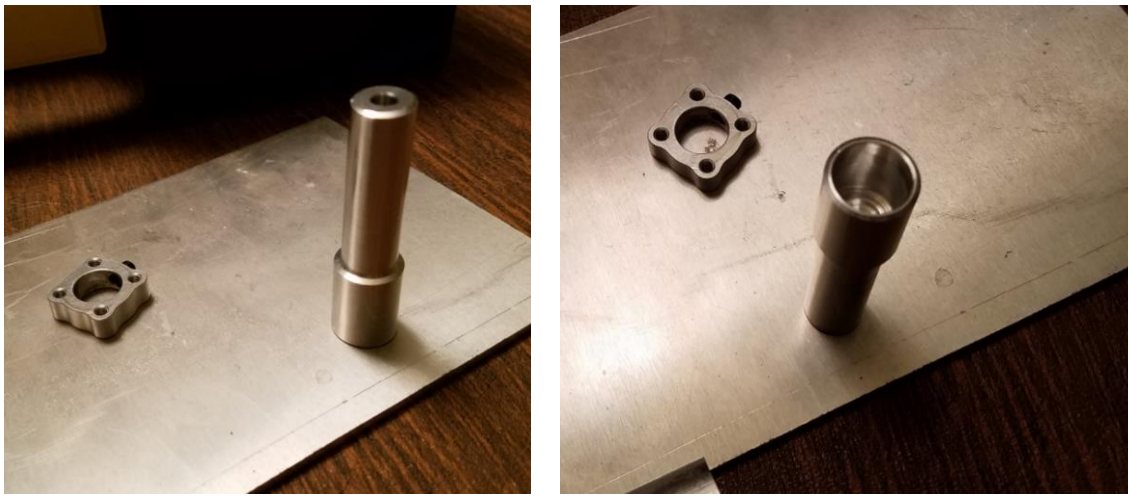


Figure 72. Prototype LIDAR spindle

This spindle is the backbone of the LIDAR system. The laser sensor itself mounts to it, the 18-tooth pulley mounts to it, and the wires flow from the laser down the middle of it, through the slip ring and out the bottom of the spindle to the microcontroller. The spindle with the slip ring mounted inside is shown in Figure 73.

Figure 73. Prototype LIDAR spindle with slip ring

The fully assembled prototype unit built as a proof-of-concept can be seen in Figure 74.

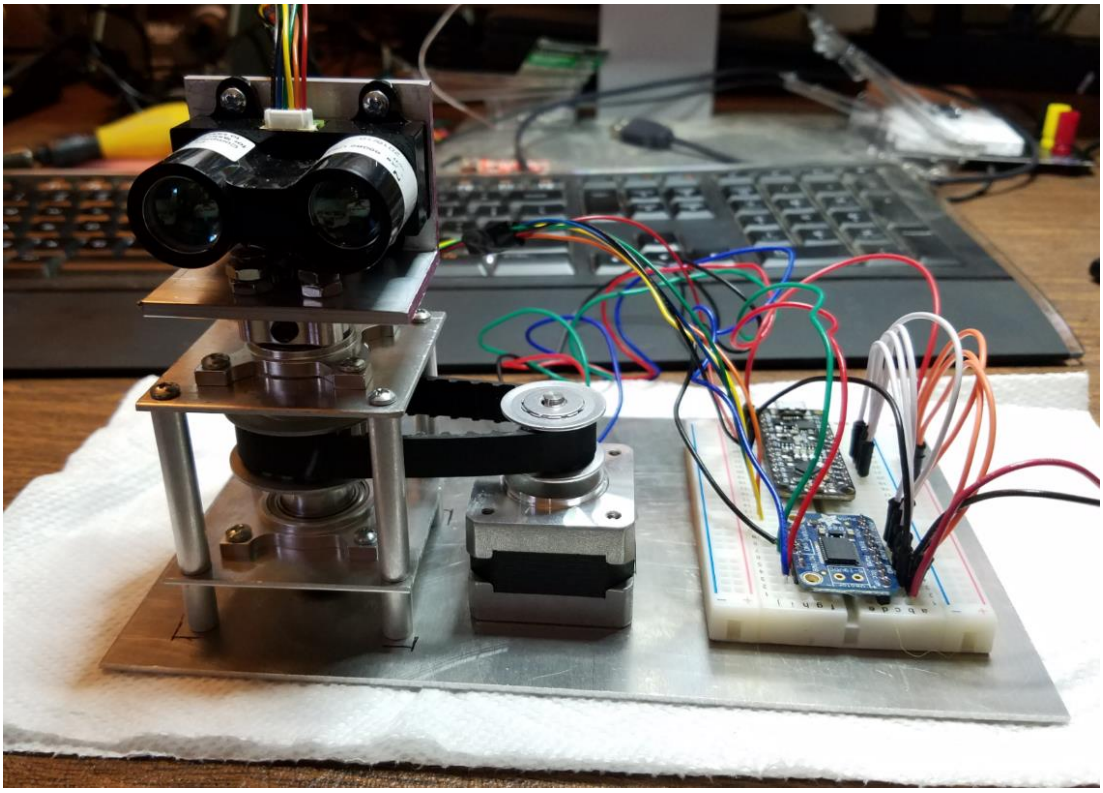


Figure 74. LIDAR prototype

Once the LIDAR prototype had been built, it allowed me to test the 360-degree scanning sweep process. The mechanical design worked well. There were various issues discovered involving the programming of both the stepper motor as well as the laser unit, which will be covered in the next sections. Now that the basic process had been confirmed, I proceeded to build out the actual housing units for the LIDARs. This involved creating a more robust case as well as a more stable and solid spindle assembly. Additionally, I needed a way to house the electronics and protect them from the dirt, dust, and vibration present in a mowing environment.

After measuring the prototype, I decided to house the unit inside a 6061 4x6" rectangular aluminum tube. The tube was 0.25" thick, which provided enough material

to drill and tap for standard 4-40 bolts, which I used to secure the top and bottom plates. The top plate was made out of 3/8" thick 4x6" 6061 aluminum bar stock while the bottom plate was made out of 1/4" thick 4x6" 6061 aluminum bar stock. I milled a recessed edge around the top and bottom plates, so they would fit securely inside the 4x6" rectangular tube. The top plate was slightly thicker in order to hold the bearing that was used to hold the laser head in place (Figure 75).

Figure 75. Creating the top plate on the mill

The laser head mount was created out of 3" solid 6061 aluminum round stock (Figure 76). This stock was machined down to size on a mini-lathe. The laser head mount was made to slip over the vertical spindle. This head mount was machined to fit inside the bearing shown in Figure 75. There was a slight lip made on the underside of the head mount, so the head mount would rest on the inner-track of the bearing (Figure 77).

Figure 76. Creating the laser head mount on the lathe

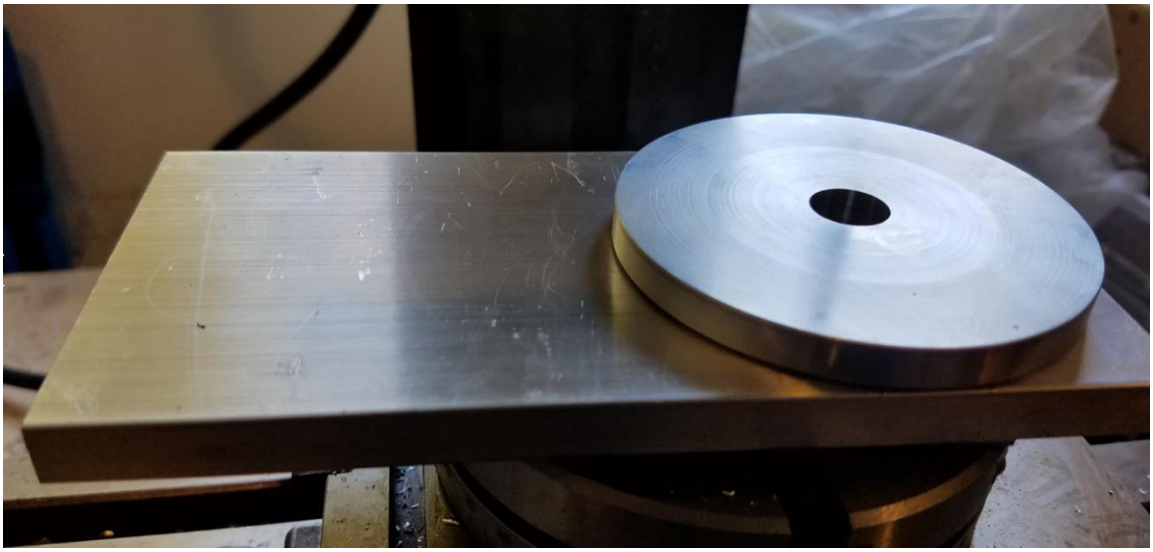


Figure 77. Laser head mounted in top plate bearing

It was imperative for the spindle assembly to line up perfectly with the top plate, so the head mount could spin freely without binding. This proved to be a difficult as the spindle assembly needed to be mounted in exactly the right position on the bottom plate in order for the spindle to be perfectly centered inside the top head plate. In order to center the spindle assembly with the top plate, I created a jig that was half the height of the 4x6" rectangular tube (Figure 78). This was done so I could assemble the pieces

together and then use the mill to pass a 1/2” end mill bit down through the top of the head plate to create a divot in the bottom plate (Figure 79a). This divot would then be used to align the spindle assembly in the correct location on the base plate (Figure 79b).

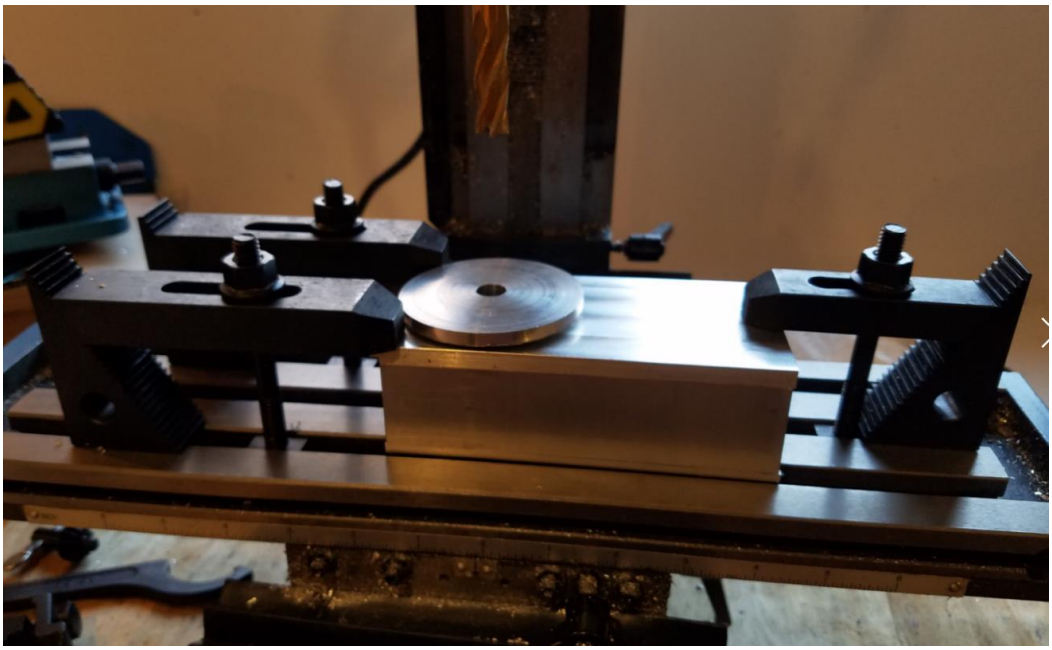
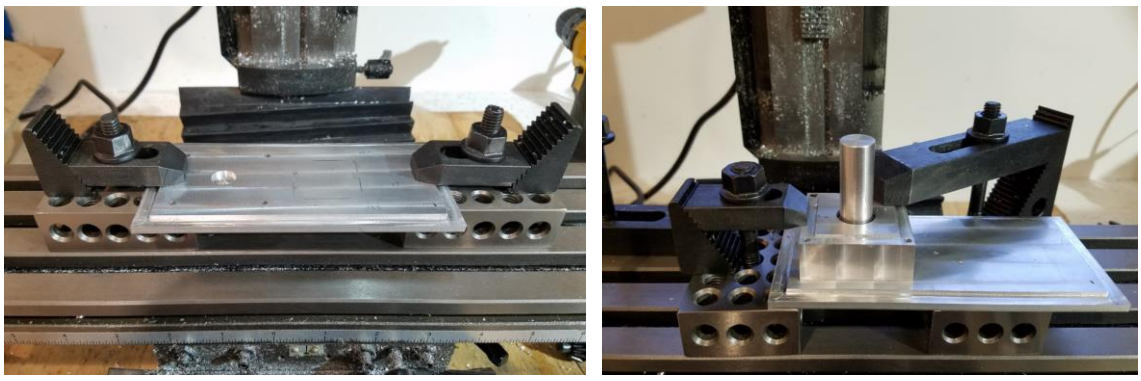


Figure 78. Alignment jig for base plate

A temporary 5/8” diameter shaft had to be created to go through the spindle assembly’s bottom bearing but then was 1/2” at the end in order to fit into the 1/2” divot created by the end mill bit. The end mill bit was 1/2” as that was the bore through the head plate.



(a) (b)

Figure 79. Base plate with alignment divot (a). Centering spindle assembly using temporary shaft with alignment divot (b).

Once everything was locked down, as shown in Figure 79b, the bolt holes for the spindle assembly could be made. Figure 80 shows the completed unit, mounted on the base plate.

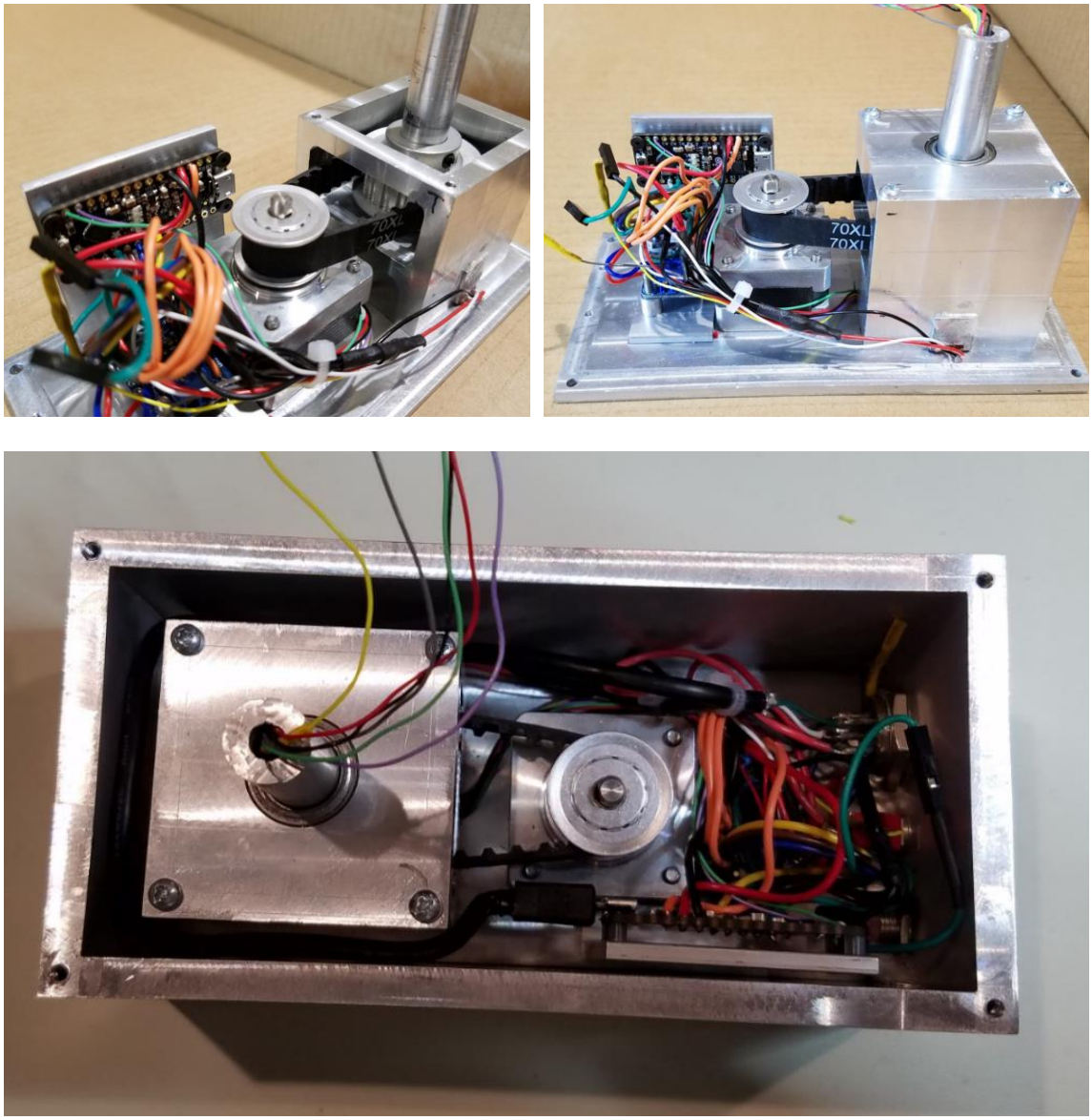


Figure 80. Inside view of completed LIDAR unit

Once the initial unit was completed, it was just a matter of repeating the process two more times to complete all three units (Figure 81).

Figure 81. Three completed LIDAR units

4.1.2 Microcontroller and Sensors

As was mentioned earlier in this chapter, there was the need to create a home position for the laser sensor. This home position can be arbitrary as long as its location remains set and can be found reliably. This home position is then designated as degree 0. In order to accomplish this goal, I utilized a magnet and Hall Switch. To mount the Hall Switch, I created a small mounting block from some raw aluminum stock (Figure 82). This mounting block was then placed inside the bottom of the spindle assembly. A magnet was then attached to the spindle assembly. The completed mounting block and magnet is shown in Figure 83. The spindle was increased in thickness moving from the prototype to the final version. This was done so that a 1/8" diameter by 1/16" deep divot

could be made in the spindle side. The magnet itself is 1/8” in diameter and 1/8” thick, leaving 1/16” of the magnet protruding out from the side of the spindle.

Figure 82. Creating the Home sensor

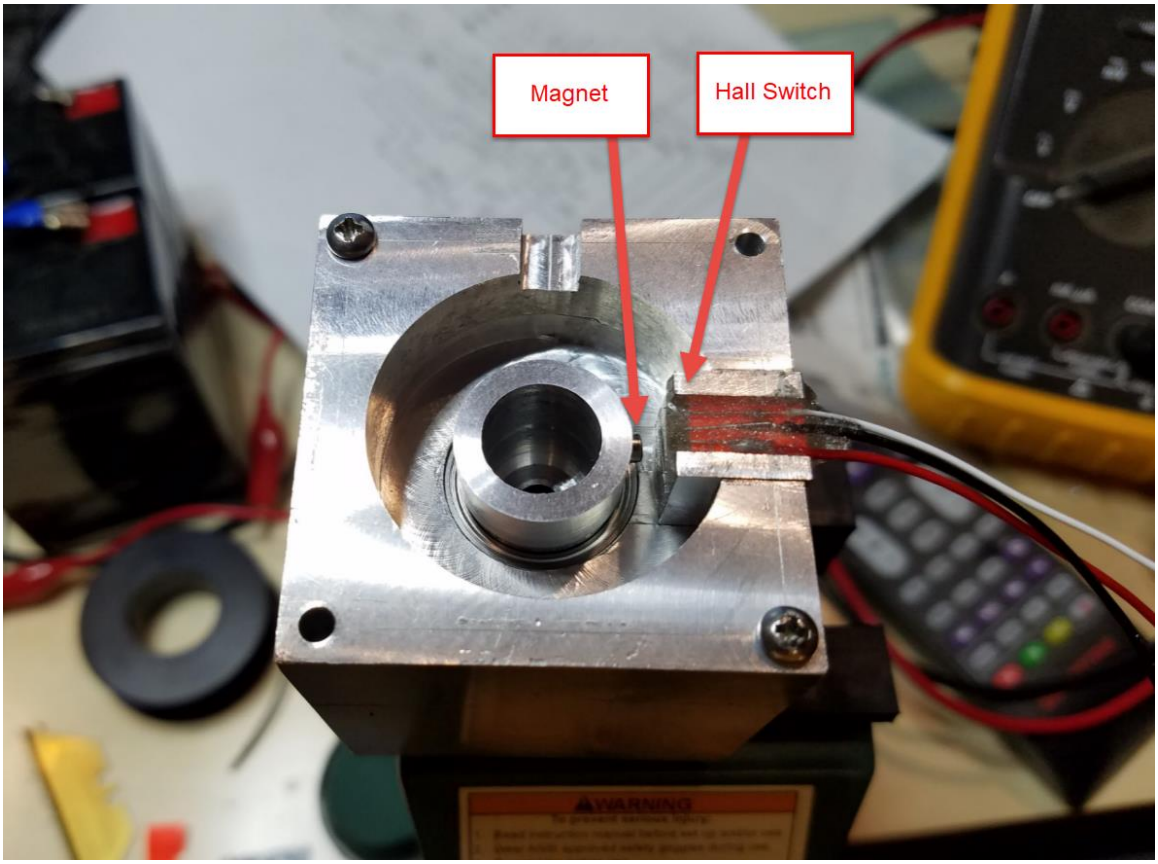
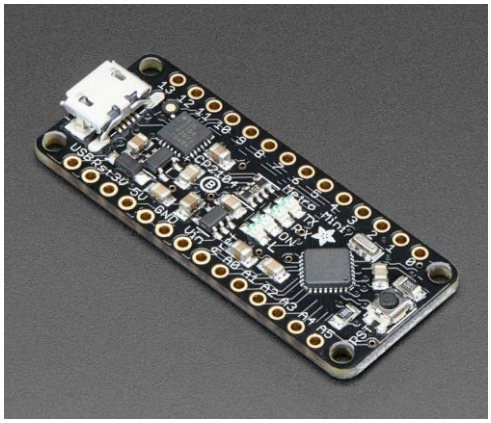


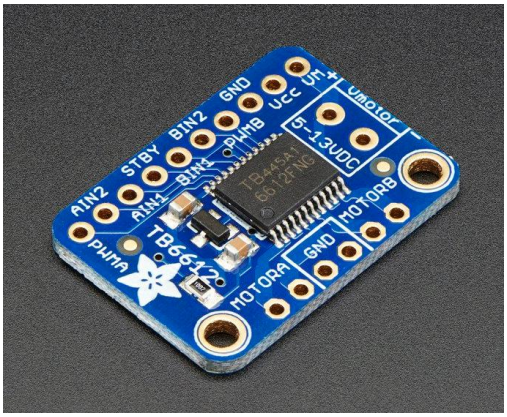
Figure 83. Magnet and Hall Switch for Home sensor

Every time the magnet passed by the Hall Switch, a pulse was sent to the microcontroller. This happened on every rotation of the spindle and therefore every rotation of the laser sensor. I tested 1,000 revolutions and the sensor reliably picked up the magnet at the exact same step on every rotation. I was not able to accurately position the Hall Switch and magnet in such a way that the Hall Switch picked up the magnet at exactly degree 0. Therefore, I built an offset into the software that I could tune in order to adjust the Home position to be forward facing. This approach worked well, especially given the different mounting positions of the three LIDAR units.

The microcontroller chosen for use in the LIDAR units was the Adafruit Metro Mini (Figure 84a). This microcontroller is based on the Atmega324 chip and is

compatible with the Arduino development environment. The advantage of this particular microcontroller is that it is inexpensive ($12.50), runs at 16 MHz and operate at 5 VDC. Many of the Arduino compatible microcontrollers run at 8 MHz and operate at 3.3 VDC. The Hall Switch also requires 5 VDC to operate, which means the microcontroller and Hall Switch can be run off the same power supply with no level shifting between voltage levels.


(a)



(b)

Figure 84. Adafruit Metro Mini (a) and Adafruit TB6612 Stepper Motor Breakout (b)

The Adafruit Metro Mini is not capable of driving a stepper motor on its own. In order to drive the stepper motor, a stepper motor controller was utilized. The TB6612 board (Adafruit, 2018e) was chosen as it is low-cost ($4.95), 5 VDC compatible, and communicates over I2C. Another advantage of the Metro Mini is that it uses USB to communicate with the host computer. This USB port is used for both programming and communication, with no need to make any hardware changes to switch between operations. The USB port for the Metro Mini was extended via a small USB cable to a 6-pin Aviation connector on the outside of the aluminum casing (Figure 85). Aviation connectors are circular connectors that screw together thereby offering higher-reliability than more simple push-on connectors. Four of the pins for this connector were used for

the USB connection while the other two pins were used for 12 VDC and Ground to power the stepper motor.

Figure 85. External 6-pin power/communication connector

The screw-on Aviation connector was chosen to provide power and USB instead of a discrete USB and power connections for two reasons. First, the screw-on connector ensures the cables won't accidentally become unplugged. Second, having one cable connection providing both power and communication enables easier cable routing and management. An on/off switch and power LED are also provided on the outside of the case.

The Adafruit Metro Mini does not have enough memory to run as a ROS node directly. This is unlike the Teensy 3.2 microcontroller used for other operations in the system, which can expose their resources directly as a ROS node. As such, in order to expose the LIDAR distancing information to the rest of the ROS environment, a method of communication between the LIDAR unit and the host system needs to be provided. The software and communication methodology are discussed in the next section.

4.2 Software

The software running on the LIDAR unit needed to perform several functions. It needed to control the Garmin Lidar Lite laser, it needed to control the stepper motor, it needed to read the Hall Switch sensor, and it needed to communicate the distance information over the USB port to the main computer. Additionally, it needed to perform all these functions in a fast, non-blocking way.

4.2.1 Stepper Motor Control

The stepper motors used in the LIDAR are bi-polar NEMA 14 stepper motors. Stepper motors are more difficult to control than normal motors due to having multiple windings used to create discrete steps. These windings need to be energized and de-energized at specific intervals in order to make the shaft step through its rotation. One of the reasons the Adafruit TB6612 breakout board was chosen for this project was that Adafruit made available a library for use in the Arduino to manage this complexity. I was able to get the library working and to successfully control the stepper motor. Unfortunately, upon further testing I discovered the library was synchronous in its operation. This meant whenever I asked the library to step the motor, it would not return control back to my program until the step had completed.

The stepper motors used in this project are 200 step motors. However, they have been reduced using 1:1.8 pulley reduction, which means it takes 360 steps to make one full revolution. With the stepper motor spinning at 2 Hz, the amount of time each step takes can be computed as follows:

1000 milliseconds / (360 steps * 2) = 1.39 milliseconds per step

This means that the program would pause for 1.39 ms while each step completes. While this worked fine with nothing else going on in the system, as soon as I tried to perform other operations, such as reading the laser sensor or transmitting data, the system would behave erratically. The erratic behavior manifested itself in skipped steps, jerky steps, missed laser scans, and incomplete data transfer over the serial port (USB). In order to resolve these issues, I did not use the stepper motor library and instead wrote my own driver to control the stepper motors.

In order to directly control the stepper motor using the Toshiba TB6612 chip, I needed to better understand how a bi-polar stepper motor worked. Motion Control Products (Motion Control Products, 2017) is a company specializing in various types of motors and controllers. They had some great information that helped me understand stepper motor theory.

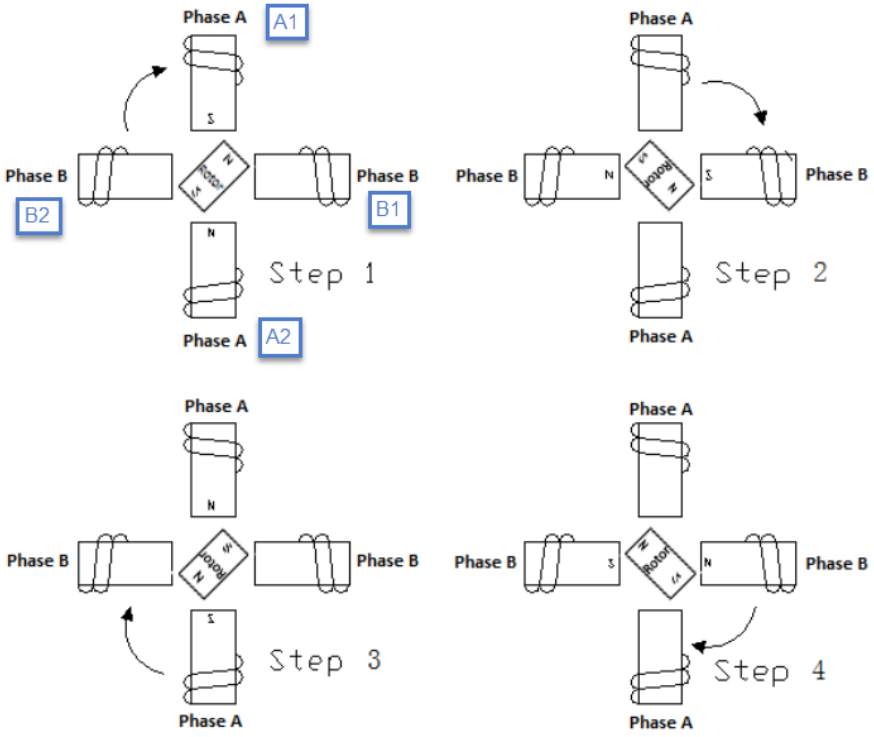


Figure 86. Bi-polar stepper motor step sequence (Motion Control Products, 2017)

Control of a bi-polar stepper motor is illustrated in Figure 86. A bi-polar stepper motor means that two phases are on at a time and it is the polarity that is switched between the phases. The two phases are labeled A and B and since each phase can have a positive (North) or negative (South) polarity, we have A1/A2 and B1/B2. This is why bi-polar stepper motors have four wires. Since opposite poles attract, in order to get the North pole of the rotor in the Step 1 position, A1 and B1 are given a negative polarity while A2 and B2 are given a positive (North) polarity. This causes the North side of the rotor to align in-between A1 and B1 while the South side of the rotor aligns between A2 and B2 (Figure 86, Step 1).

With the rotor in the Step 1 position, in order to move the rotor to the Step 2 position, the North side of the rotor needs to be pulled down, so it aligns between B1 and A2 (Figure 86, Step 2). B1 and A2 are therefore switched to have a negative polarity while A1 and B2 are switched to have a positive polarity. This causes the rotor's North side to align in between B1 and A2 while the South side aligns between B2 and A1. Now with the rotor in the Step 2 position, in order to move to the Step 3 position, the North side of the rotor needs to be aligned between A2 and B2 while the South side of the rotor needs to be aligned between A1 and B1 (Figure 86, Step 3). Therefore, A2 and B2 become Negative (South poles) while A1 and B1 become Positive (North poles). Step 4 is the last step position. Moving the rotor into this position from the Step 3 position means pulling the North side of the rotor between A1 and B2 while the South side aligns between B1 and A2 (Figure 86, Step 4). To do so, A1 and B2 become Negative (South poles) while B1 and A2 become Positive (North poles). At this point, the process can be repeated to move the rotor through another four steps.

This overall stepping process is illustrated with associated stepper wire states in Table 9. It is important to note that due to the way the stepper motor is wired, in order to obtain a negative polarity, the associated stepper motor wire must be driven high. Likewise, in order to obtain a positive polarity, the associated stepper motor wire must be pulled low.

Table 9. Stepper motor coil sequence. Pulling a wire high results in Negative polarity (-) while pulling a wire low results in Positive polarity (+), at the motor coil.

Step	Wire 1 (A1)	Wire 2 (A2)	Wire 3 (B1)	Wire 4 (B2)
1	High (-)	Low (+)	High (-)	Low (+)
2	Low (+)	High (-)	High (-)	Low (+)
3	Low (+)	High (-)	Low (+)	High (-)
4	High (-)	Low (+)	Low (+)	High (-)

A Finite State Machine (FSM) is ideal for representing the various states and transitions for the bi-polar stepper motor. An FSM can be used to control the transitions from one state to the next state as well as describe the starting state required to move to the next state. The state diagram shown in Figure 87 represents these state transitions.

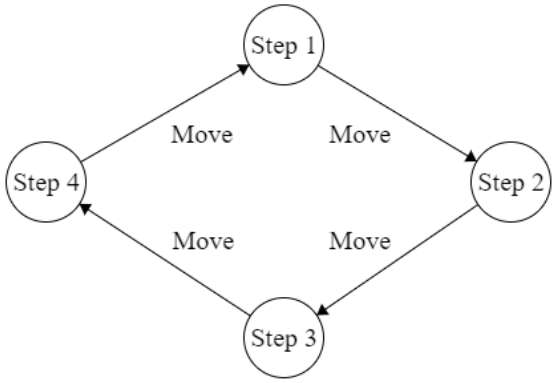


Figure 87. Stepper motor state diagram

Every command to Move causes a state transition from the current step to the next step. The diagram indicates that you must transition from Step 1 to Step 2 to Step 3 to Step 4 and then back to Step 1. You cannot transition from Step 1 to Step 3 or from Step 2 to Step 4. The sequence must be followed in order.

The state machine describes how to make the stepper motor move. The next step was to control the RPM of the motor. Stepper motor speed is directly related to how much time elapses between steps. This is really the heart of where a synchronous versus asynchronous design diverge. In a synchronous stepper motor program, such as the one I initially tried, the program would simply pause before moving to the next step. Rather, I needed my stepper motor program to run asynchronously so it would not block the rest of the program from running as it waited to move from one step to the next. Other operations needed to be performed concurrently with the stepping process, such as reading data from the laser sensor. In order to create a non-blocking stepper controller, I made use of a similar technique used for the throttle control. Namely, I made use of the micros function present in the Arduino environment.

The code shown in Figure 88 represents the basic control structure used to implement non-blocking control of the bi-polar stepper motors. This code is part of the main processing loop for the LIDAR units, which means it runs continuously but doesn't block and prevent the rest of the code in the main loop from running.

```
unsigned long currentMicros = micros();

if (currentMicros - previousMicros > stepDelay) {
    previousMicros = currentMicros;

    if (stepState == STEP1) {
        digitalWrite(inA1, LOW);
        digitalWrite(inA2, HIGH);
        digitalWrite(inB1, HIGH);
        digitalWrite(inB2, LOW);
        stepState = STEP2;
        currentStep++;

    } else if (stepState == STEP2) {
        digitalWrite(inA1, LOW);
        digitalWrite(inA2, HIGH);
        digitalWrite(inB1, LOW);
        digitalWrite(inB2, HIGH);
        stepState = STEP3;
        currentStep++;

    } else if (stepState == STEP3) {
        digitalWrite(inA1, HIGH);
        digitalWrite(inA2, LOW);
        digitalWrite(inB1, LOW);
        digitalWrite(inB2, HIGH);
        stepState = STEP4;
        currentStep++;

    } else if (stepState == STEP4) {
        digitalWrite(inA1, HIGH);
        digitalWrite(inA2, LOW);
        digitalWrite(inB1, HIGH);
        digitalWrite(inB2, LOW);
        stepState = STEP1;
        currentStep++;

    }
}
```

Figure 88. Non-blocking Arduino stepper motor control

The micros function returns the number of microseconds since the program started running. Upon initial startup, the previousMicros variable will be zero. The stepDelay variable contains the number of microseconds to wait between steps. The stepDelay value represents the longest period of time available to perform an operation. The next step must occur after stepDelay microseconds in order to ensure smooth stepper motor operation. Given the short amount of time this variable represents, the accumulated amount of blocking work that needed to be performed had to fit within this

period. This means the maximum rotational speed is environmentally specific as it is tied to the amount of work that needs to be performed for each rotation.

The size of an unsigned long for an Adafruit Metro board is 32 bits. This means the largest number an unsigned long can hold is 2^32 - 1 or 4,294,967,295. As mentioned, currentMicros is an unsigned long variable representing the number of microseconds that have elapsed since the Arduino was started. As such, it will roll over after 4,294,967,295 microseconds, or 71.58 minutes. It is entirely feasible for the mower to operate longer than this length of time when in operation. When I mow my backyard, it takes two hours. The code shown in Figure 88 was written to handle the possibility of overflowing the variables used to store micros. The reason being is that I am comparing durations, not timestamps. So long as the duration being compared is short enough to fit within an unsigned long on my 32-bit platform, I can use this technique without having to check for an overflow. This works because the computed duration itself also rolls over. As an example, assume the previousMicros is 100 microseconds from the rollover (4,294,967,196) and the currentMicros has rolled over and is now 100 microseconds after the rollover (100). When the subtraction occurs (currentMicros – previousMicros) the result would be negative in signed arithmetic, but since the variables are unsigned long types, the result is also an unsigned long and therefore rolls over to become 200 in unsigned arithmetic.

Given the small amount of memory available in the Adafruit Metro microcontroller, all data gathered for each rotation needs to be fully transmitted to the main host computer during each rotation. The time it takes to transmit each rotation's data over the serial USB port is dependent on how much data is being transmitted and the

baud rate. The Adafruit Metro board has a CP2104 chip working as a USB to Serial Bridge with a maximum baud rate of 2 Mbps (Silicon Labs, 2017). However, in my testing, the electromagnetic noise generated by the mower's engine and spinning blades allowed for a maximum baud rate of 115,200 bps. Going with higher baud rates introduced errors when transmitting the data from the LIDAR units to the main computer. This maximum baud rate was determined empirically after additional shielding was added to the USB cables to help reduce noise from the engine and spinning blades. In my specific environment, 2 revolutions per second was the maximum rotational speed I could achieve without pausing during each rotation given the amount of data being transmitted and the baud rate used. The stepDelay value was computed as follows: 1,000,000 microseconds per second / 720 steps per second = ~1,388 microseconds per step. Using a value of 1388 for the stepDelay thus yielded 2 revolutions per second.

Initially, I was not able to get 2 revolutions per second using a value of 1388 for stepDelay. Instead, during my testing, a stepDelay value of 1250 resulted in 2 revolutions per second (720 steps per second). The difference between the calculated value of 1388 microseconds and the empirical value of 1250 microseconds meant there were close to 100 milliseconds of time unaccounted for in the program every second:

(1388µs/step-1250µs/step) * 720 steps/second = 99,360µs/second

After spending time working through all the code in the system, I eventually discovered I had left logging turned on in the LIDAR lite library during previous testing. This logging was writing data out to a UART. Once this logging was turned off, the calculated stepDelay value of 1388 resulted in 2 revolutions per second, as expected.

I did consider using one of the several built-in timers available on the Adafruit Metro board, along with an associated interrupt handler, to process the state machine logic. This approach would have allowed me to put the stepper motor state machine logic into a separate interrupt handler that would then have been called based on a predefined clock rate. However, ultimately, I decided against this approach as it would have tied me more closely to the hardware used in the Adafruit Metro as this approach is based on the clock rate of the board's timers. This means it would require more work to change the code to another microcontroller. During testing, I did change microcontrollers several times for troubleshooting purposes. Not having to research how the timers were used between microcontrollers and recompute clock frequencies was a definite time-saver. Since there are 360 steps per revolution, and there are 2 revolutions per second, this means there are 900,000 microseconds in my non-blocking code. Since there are 1,000,000 microseconds in 1 second, there are 100,000 microseconds unaccounted for in the above code. This time is likely due to working with other various hardware ports such as the digital input for the Hall Switch home sensor, the I2C communication for the laser module, and the USB Serial communication with the main computer.

4.2.2 Recording Distance Readings

The Garmin laser module has an Arduino library available for reading distance information from the Lidar Lite v3 over the I2C interface. Unfortunately, this library also turned out to be synchronous and therefore, blocking. Fortunately, I was able to locate an open-source library called Lidar Lite v3 Enhanced (Paques, 2017) that performed better than the original library from Garmin, was non-blocking, and also supported multiple laser modules. I only needed to use one laser module for each LIDAR unit, so I did not

make use of this last feature. However, the fact that it could make use of the higher I2C speeds was a bonus. The LIDAR library operates in background. Through each iteration of my program's main control loop, the LidarController spinOnce function is called. This function updates any current distance readings as well as the internal state machine for any connected laser sensor modules.

Combining the non-blocking LIDAR library with my stepper motor controller was straightforward. Every time I received a laser scan, I retrieved the current step position and used that as an index into an array to store the distance reading. The array contained 360 int values, one for each step. Due to the hardware design of the LIDAR unit, each step corresponded to one degree. Once I completed 360 steps, I knew I had completed 1 full revolution. At this point, I transmitted the distance data stored in the array to the main computer over the USB serial connection and then zeroed the array.

One additional item I had to account for was the ambient light bias correction. According to the Lidar Lite v3 operator's manual, when taking a measurement, the sensor first performs a received bias correction routine, which corrects for changes in the ambient light levels. The manual does not go into details as to what this routine does behind the scenes, but it does state that running this routine allows for maximum sensitivity of the laser sensor. However, I found this routine could significantly impact the speed at which the sensor could operate. Running this routine before every measurement could slow the sensor down 10x or more. Instead of running the routine on every scan, I found through empirical testing under various lighting conditions the sensor remained unaffected by changing light conditions if I re-sampled the ambient light every 100 measurements.

4.2.3 Serial Communication Protocol

The LIDAR units are running an Adafruit Metro Mini microcontroller. These microcontrollers do not have enough memory to run directly as a ROS node. This is both an advantage as well as a disadvantage. The disadvantage is that rather than directly exposing the sensor readings to the ROS environment, they need to be sent over the USB serial connection to another program and from there broadcasted out to the ROS environment. The advantage to this approach is that the LIDAR units can be used inside environments not running ROS—such as Microsoft Windows. This proved useful for initial testing and debugging of the LIDAR units as Microsoft Windows was my primary development environment.

The challenge I ran into sending the data over a serial connection had to do with the amount of data coupled with the speed of the microcontroller. There was very little time to transmit the data while continuing to both step and obtain laser scans. The Arduino environment provided functions for transmitting text-based data through the serial port. The Arduino print functions took a string of data, converted it to ASCII, and then sent this data over the serial port. Since the data was readable in this format, it was useful for debugging but proved too slow to work effectively. Using this approach led to significant pauses on each half-revolution as the LIDAR unit transmitted the data to the main computer.

A new more compact encoding scheme was created to enable transmitting the data to the main computer. The data to be transmitted were distance readings in centimeters organized by degree. There would be 360 degrees for each complete revolution. The laser module was capable of producing distance readings up to 40

meters, which is 4000 centimeters. The data could not simply be sent as raw integer values because there would be no frame of reference. This means that when the data started arriving at the host computer's serial port, without some sort of framing, I would not know what values were associated with what degree. Any attempt to introduce framing without an encoding was problematic as well since any value picked for coding could show up in the data. Therefore, the data required encoding.

I chose to start each frame of data with the Start Transmission STX (0x02) character and end the frame of data with End Transmission ETX (0x03) character. I then encoded the integer values representing distance as three ASCII characters representing the value in hexadecimal. The maximum value that can be represented in three ASCII characters representing the value in hexadecimal (FFF) is 4095, which is slightly over the upper-value limit requirement of 4,000. The degree is represented by the distance location in the frame. Since each distance location is 3 bytes, the degree is represented by the zero-based distance offset inside the frame. The degrees range from degree 0 up to and including degree 359, corresponding directly with the offset in the array. When the host program starts up, there may already be data in the serial buffer. This data is discarded until an STX character is found and a full frame can be made.

Converting the packet data from a binary integer to an ASCII representation of the hexadecimal value is straightforward using the sprintf function. Likewise, reading the data on the host as a hexadecimal string and converting it to a long integer can be done using the strtol C function. The atoi C function was avoided due to it lacking a mechanism for reporting invalid output. Given that communication is Serial but is occurring over a USB port, there is no need for additional error handling. According to

the USB specification (usb.org, n.d.), USB has built-in Cyclical Redundancy Checksums (CRCs) for error detection in data packets. Therefore, I did not include any error checking in the packet structure. Based on my testing, I was able to successfully send data from the LIDAR unit to the main computer at 2 Hz with a resolution of 1 degree and no stuttering of the process. Attempting to increase the scan rate above 2 Hz resulted in the LIDAR pausing while transmissions were being sent. At the maximum current rate of 2 Hz, there are 720 distances readings being sent to the main computer every second.

4.2.4 Custom ROS/Ubuntu Serial Service

The overall goal of the LIDAR unit was to provide the ROS navigation stack with a LaserScan message. The ROS navigation stack used the LaserScan messages to generate maps of the robot's environment. I then used these maps to perform localization (chapter 5) and path planning (chapter 7). The ROS navigation stack can only subscribe to one LaserScan publisher, which meant the ROS service I created needed to merge the three scans together in order to output a single combined LaserScan message. Additionally, since the front two LIDAR units cover the same 180-degree area directly in front of the mower, these units were interleaved to provide additional LaserScan messages.

These additional LaserScan messages were used for obstacle detection at a rate of 4 Hz. These additional messages were not used by the navigation stack of ROS for map building, but instead by the watchdog service I created to prevent the mower from hitting objects that quickly made their way in front of the mower. The watchdog service monitors incoming LaserScan messages on the /FrontRightLaser and /FrontLeftLaser Topics for any objects 2 feet outside from the mower's stopping distance. The watchdog

service subscribes to the /imu Topic (Chapter 5) to get the speed of the mower. The stopping distance is calculated using the formula:

distance = velocity^2 / (2 * (coefficient of friction) * (gravitational acceleration))

Gravitational acceleration is constant at 9.80 m/s^2. The watchdog service has defined the various coefficient of friction values in Table 10. However, it currently uses the default value for GRASS.

Table 10. Watchdog Service coefficient of friction constants (Cenek, 2016)

Surface Type	Coefficient of Friction
GRAVEL	0.35
GRASS	0.20
DRY_ASPHALT	0.65
WET_ASPHALT	0.50
DRY_CONCRETE	0.75
WET_CONCRETE	0.60
SNOW	0.22
ICE	0.12

In order to turn the raw data coming from the custom LIDAR units into a LaserScan message requires the presence of a ROS node. This ROS node receives serial data through the USB port from the LIDAR unit, parses it and then reposts the data as a LaserScan message. Table 11 shows the data present in a ROS LaserScan message (ROS, 2012a).

Table 11. ROS LaserScan Message (ROS, 2012a).

Name	Type	Description
header	Header	The header includes a sequence, a timestamp that should be the acquisition time of the first ray in the scan. The header frame_id is the frame upon which the angles are measured around. The positive Z axis (counterclockwise, if Z is up), with zero angle being forward along the x axis
angle_min	float32	Start angle of the scan (radians)
angle_max	float32	End angle of the scan (radians)
angle_increment	float32	Angular distance between measurements (radians)
time_increment	float32	Time between measurements (seconds)
scan_time	float32	Time between scans (seconds)
range_min	float32	Minimum range value (meters)
range_max	float32	Maximum range value (meters)
ranges	float32[]	Range data (meters)
intensities	float32[]	Intensity data if available (device specific units)

The LIDAR units used in this project take 360 degree sweeps of the area. However, given the mounting position of the units, part of the scan area of each laser is occluded by the mower itself. That said, the LaserScan message output for purposes of mapping contained the merged laser scan data and has an angle minimum of 0, an angle maximum of 359 (6.28319 radians), and an angle increment of 1 degree (0.0174533 radians). The minimum range is 0 and the maximum range is 40 meters for all LaserScan messages used in this project. These values are defined as constants for the LaserScan message.

Prior to testing the combined LaserScan outputs, I connected one of the LIDAR units to the system and published its data to a test Topic I created called /scan. This allowed me to ensure that all parts of the system were functioning as expected and provided an easy way to debug the system. To perform this initial test, I put one of the LIDAR units into a large box (Figure 89). Then started my ROS LIDAR node.

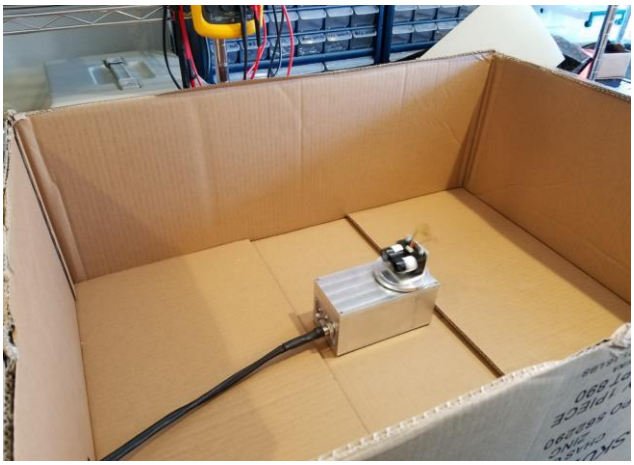


Figure 89. Single LIDAR unit testing

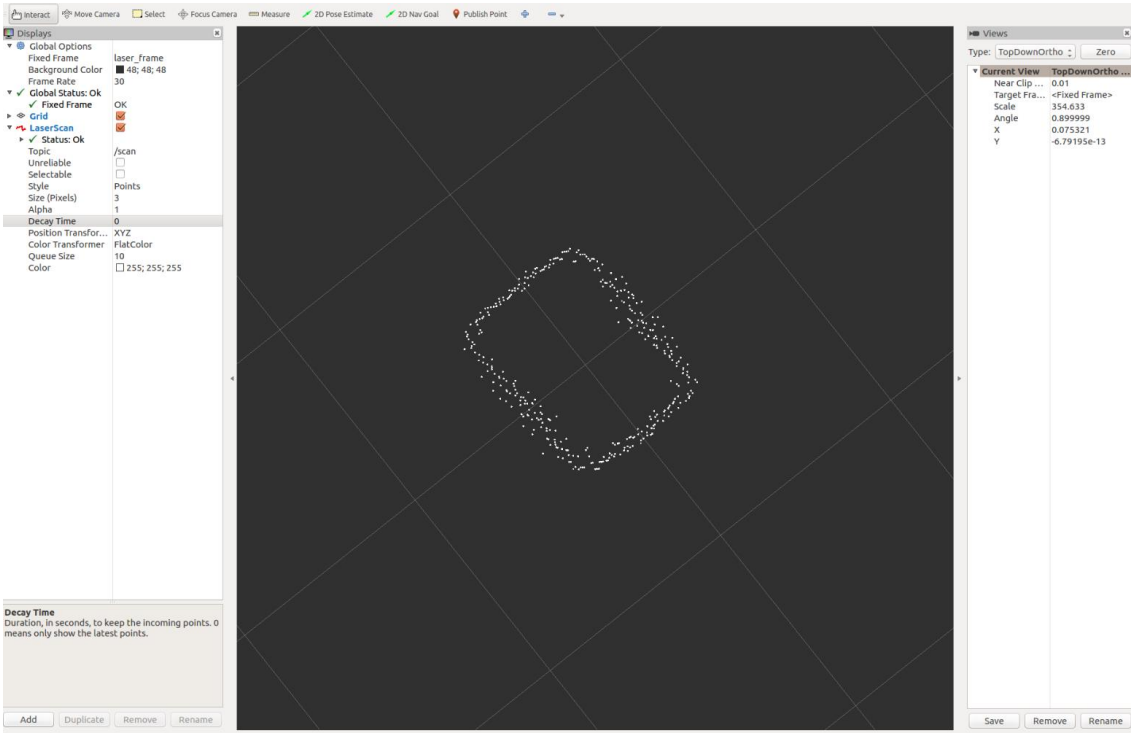


Figure 90. RVIZ connected to the LIDAR /scan topic

Once my ROS LIDAR node was running, I was able to start the ROS Visualizer program (RVIZ) and connect it to the /scan test Topic I created (Figure 90). The RVIZ program plots the location of all range points on a fixed map. The Garmin manual indicates there is some non-linearity of the scans less than 1 meter. However, as can be seen in the screen shot, it still provides a good representation of the boxed in area. Each grid cell on the map represents 1 meter.

Next, I opened the side of the box facing me to widen the scan area. I also moved the scanner away from the back edge of the box. Now that the scanner was further from this edge, there were less incongruities in the data. A picture of this process can be seen in Figure 91.

Figure 91. Single LIDAR unit testing in a wider area.

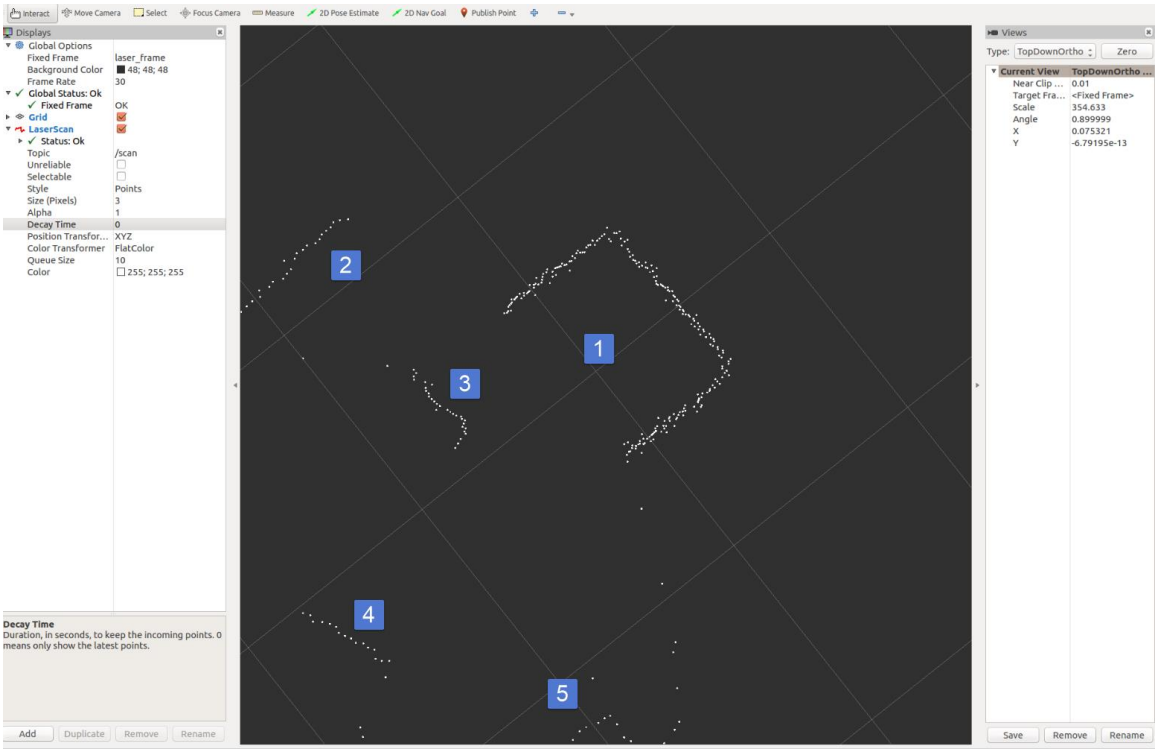


Figure 92. RVIZ connected to the LIDAR /scan topic with the wider scan area

ROS is visualizing the expanded area using RVIZ in Figure 92. The area marked 1 is the original box area with the LIDAR unit present. The area marked 2 is a computer monitor. The area marked 3 is where I was standing when I took the screen shot. The area marked 4 is the back of a chair. The area marked 5 is the corner of a tool chest. I was impressed with how well the data could represent the area.

Once the ROS node was functioning well with a single LIDAR unit, the time came to merge the individual LIDAR units together and publish the data as a single LaserScan message. The Topic I created to publish this data was called /VirtualLaser. Merging the laser data proved more difficult than initially envisioned. Initially, I expected to simply take the relevant degrees from each of the scanner range arrays and

put them together into a new array. I would then take an average of the distances from the overlapping sections of the front lasers (Figure 70). The actual process ended up being more difficult because the frame of reference for each LIDAR unit was different. In order to combine the distance values, I first had to normalize the frame of reference. The ability to transform sensor data relative to the robot base is a key feature of the ROS environment. It is able to correlate data from multiple sensor streams in both time and space, relative to any frame of reference in the system. This functionality is simply referred to as tf. I used the tf package to transform the three LIDAR units, so they would all report readings relative to the center of the mobile platform. This can be seen in Figure 93 for the indoor platform I created for testing purposes.

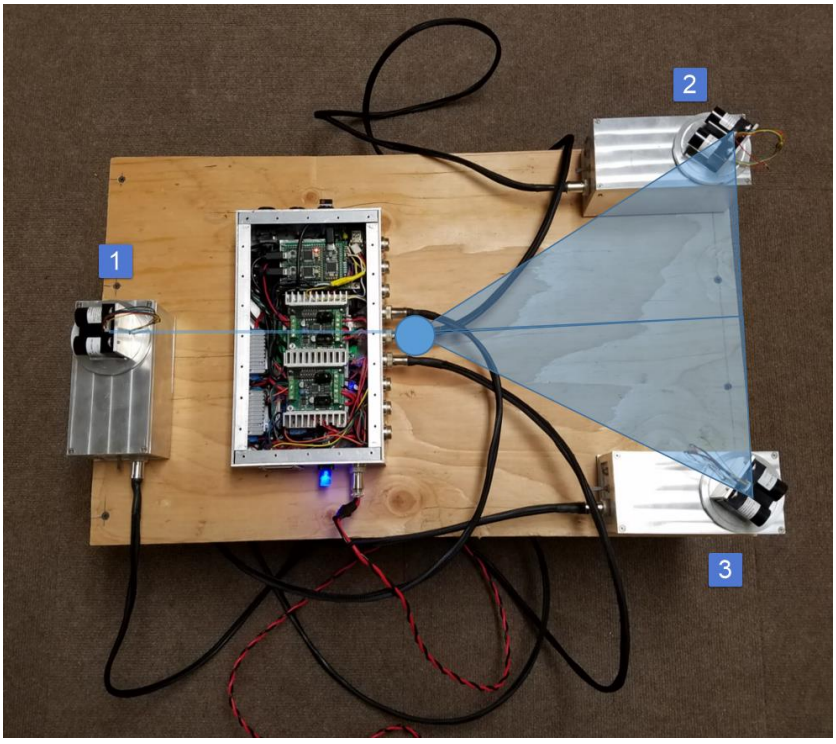


Figure 93. Merged LIDAR test platform showing relative offsets to center

The indoor test platform was 18” off the floor and with dimensions that is of the same proportion as the mower itself. The main computer is located in between the three

LIDAR units. The center of the base is marked with a blue circle. Once the distances from the three LIDAR units were merged together, it was then published as a Topic called /VirtualLaser. Figure 94 shows the merged data in RVIZ along with images of the area scanned.

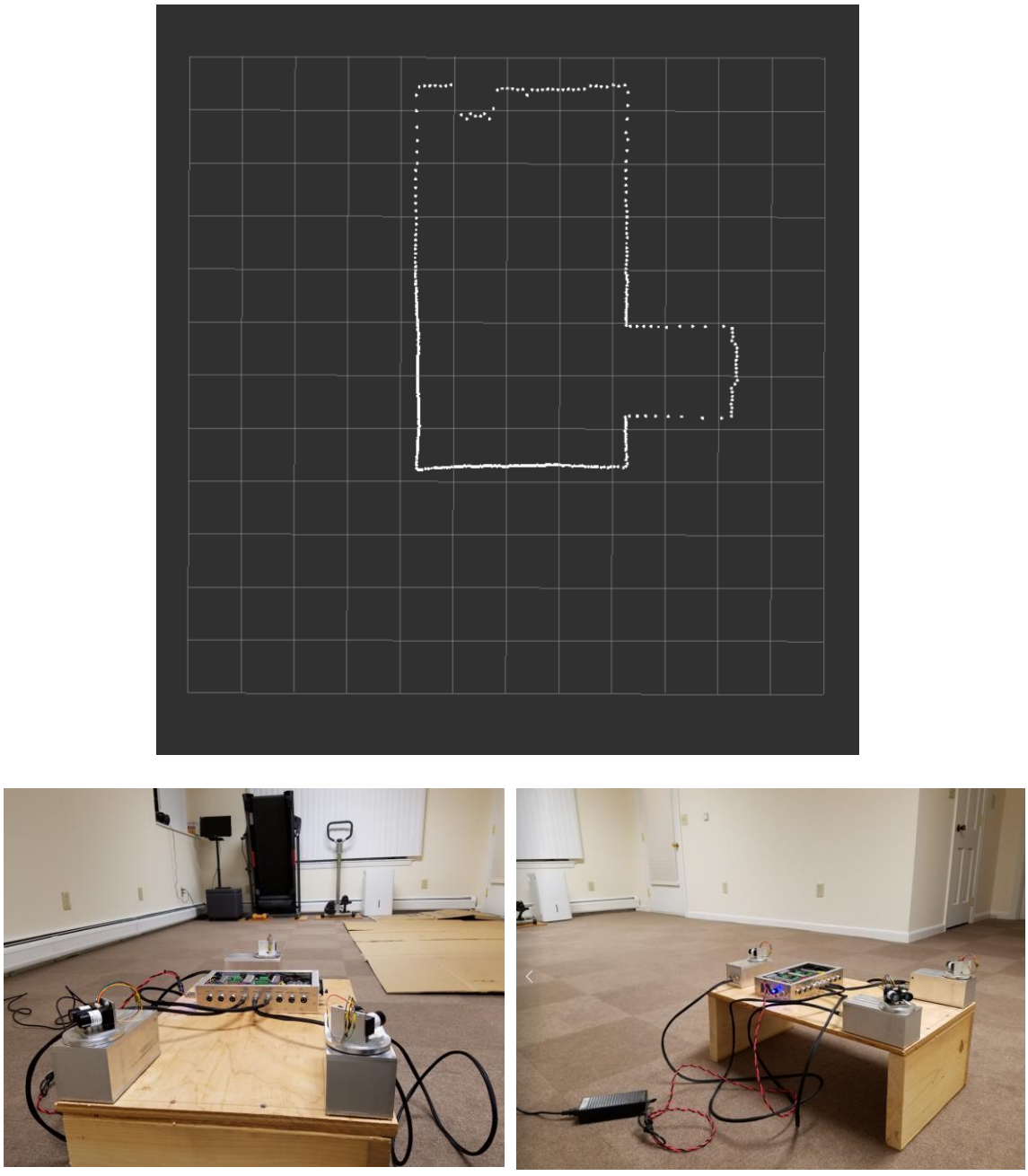


Figure 94. Merged LIDAR testing

As can be seen from the images, the merged LIDAR scans were successfully able to map the room, removing occlusions formed from scanning overlaps.

4.3 Summary

This chapter covered the unique custom LIDAR units that were built for the ASMO project. This includes details on the Lidar Lite v3 sensor module from Garmin as well as details on the Adafruit Metro Mini microcontroller and the Hall Switch home sensor. The use of the stepper motor was also covered providing details on how it was used to precisely determine where in the rotation the laser was currently distancing. In addition to the hardware covered, the software running on the microcontroller was also reviewed as well as how the microcontroller communicated with the main computer. These LIDAR units were used as the primary source of environmental knowledge for purposes of map making, localization, and planning covered in subsequent chapters.

Chapter 5

Sensor Fusion and the Odometry System

The laser distance system discussed in the last chapter provides ASMO with the means of producing a map of its operating area. This chapter discusses the sensors and software necessary to provide ASMO with a means of locating itself inside this map. A typical way of performing localization is known as dead-reckoning and can be done simply with wheel encoders. A wheel encoder is a sensor mechanism that outputs a fixed number of pulses for each turn of a wheel. Dead-reckoning is done by counting the number of pulses to find the number of times a wheel has turned and then computing the distance traveled by multiplying the circumference of the wheel by the number of turns the wheel has made. Knowing a starting location and distance travelled allows for the location to be tracked. The problem with this approach is that errors accumulate over time due to inaccuracies of the wheel encoder introduced by wheel slip.

The inertial management unit (IMU) sensor system used in this project combines sensor data from a gyroscope, an accelerometer, and a magnetometer together in order to output an absolute orientation of the sensor to the Earth's magnetic North pole. While all the discrete sensor information is available for use in this sensor, I am making use of the IMU's absolute orientation output, which represents a fusion of the discrete sensors. I fused the IMU heading data with the wheel encoder data to help reduce errors due to wheel slip. In the ASMO project, various fusion techniques were explored including the use of an Extended Kalman Filter, which is a way of combining non-linear data. As this chapter will demonstrate, the fused data is much more accurate than then wheel encoder

data alone. The fused data is then made available to the rest of the system in the form of odometry messages.

5.1 Hardware

There are several aspects to the hardware used for sensor fusion and localization. The custom lasers were covered in Chapter 4. The IMU and GPS are contained inside the Radio Box (Figure 95). This Radio Box contains the MoteinoMEGA microcontroller with the 900 MHz radio transceiver, which is used to receive radio signals from the remote-control unit. In addition, the MoteinoMEGA microcontroller also interfaces with the GPS and IMU (Figure 96). I am currently using a breadboard to connect the components inside the Radio Box. This was done for expediency during development and testing—it will be converted over to a more permanent solution, using soldered connections, in the future. A Teensy 3.2 microcontroller inside the main computer box is responsible decoding signals from the wheel encoders. The wheel encoder sensors themselves are located outside the Radio Box near the wheels. These various components are outlined in the sections below.

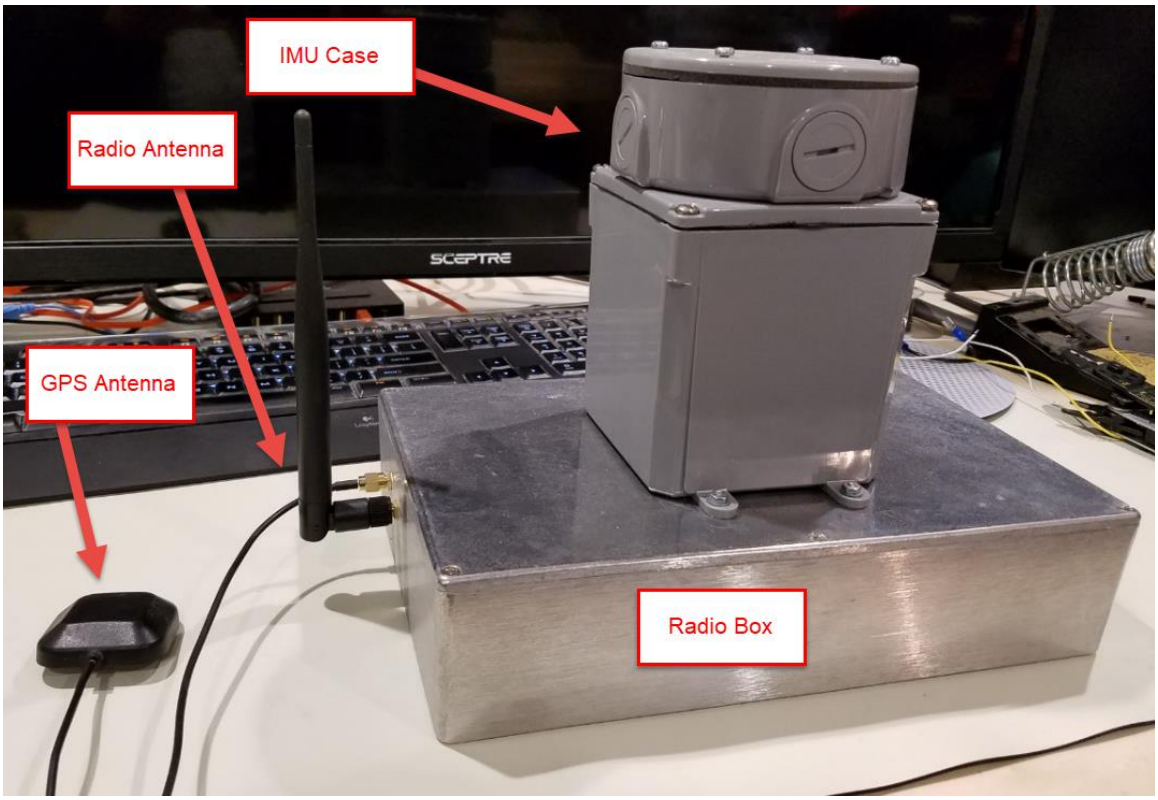


Figure 95. Radio Box containing the remote-control base, GPS, and IMU

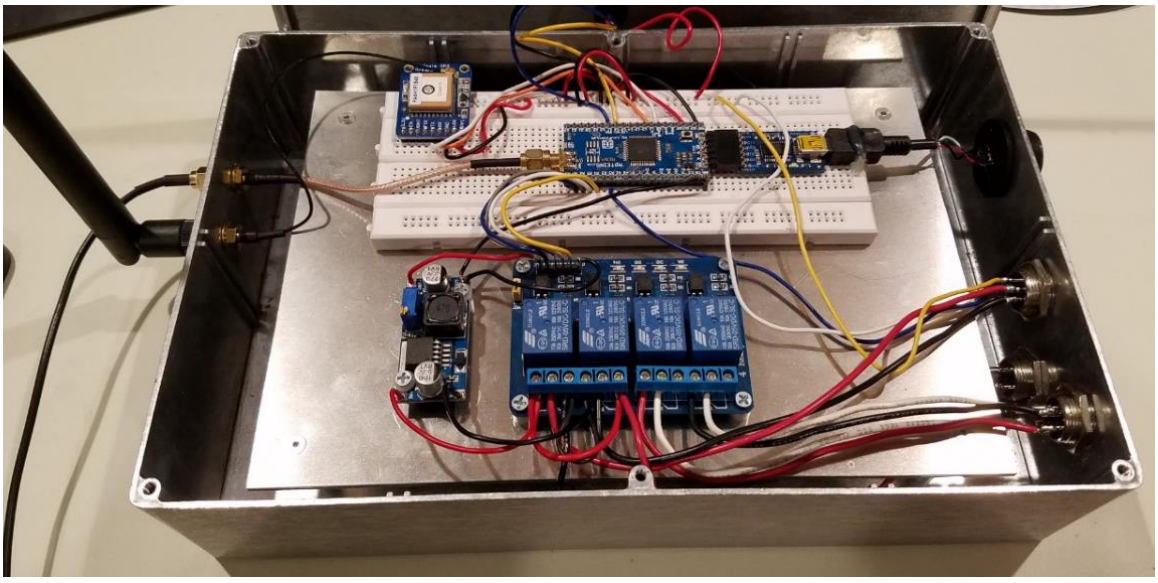


Figure 96. Inside view of the Radio Box

5.1.1 Wheel Encoders

The task of a wheel encoder is to provide a certain number of pulses per revolution of a wheel. In this way, basic geometry can be used to take the diameter of the wheel and compute the circumference (πd), which equates to the distance travelled. A typical way of generating these pulses is to mount an infrared emitter and receiver on either side of a disk with slits in it. As the slits pass in front of the infrared emitter, it breaks the beam and thereby generates a pulse that can be counted. An enhancement to this technique is the quadrature encoder, which also allows for the direction of wheel rotation to be known. A quadrature encoder disk is shown in Figure 97.

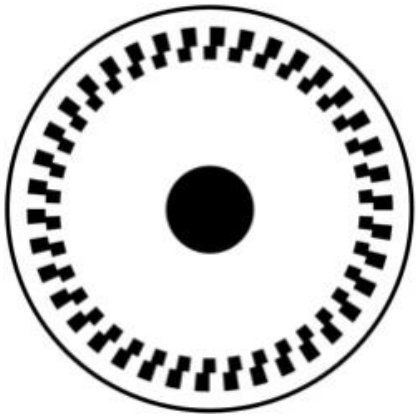


Figure 97. Optical quadrature encoder disk (Phidgets, n.d.)

The quadrature encoder works by offsetting sensors, so the pulses are generated 90 degrees out of phase. Using this approach, a microcontroller reading the pulses knows which way the encoder disk is turning based on which sensor outputs a pulse first. This process can be seen in Figure 98.

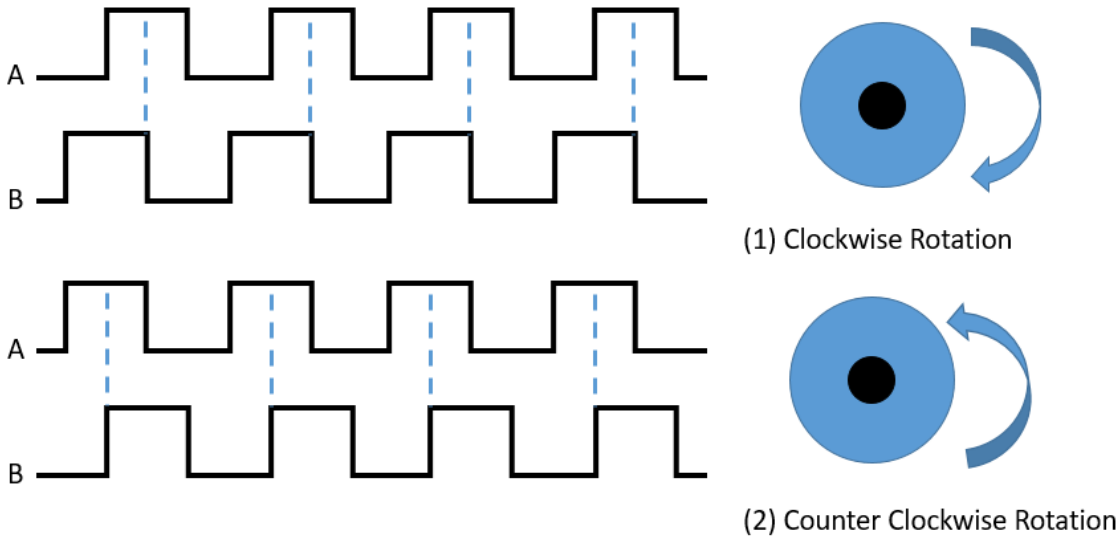


Figure 98. Encoder pulses and wheel rotation

Figure 98(1) shows channel A leading channel B, which signifies the wheel is turning in a clockwise rotation. Figure 98(2) shows channel B leading channel A, which signifies the wheel is turning in a counter-clockwise rotation.

These optical encoder systems work well in environments that are clean and relatively dust-free. However, mowing environments are inherently dirty and dusty. The slits these optical systems rely on for light to be intermittently blocked can become unintentionally blocked by dust and other grime from a dirty environment. As such, optical encoders are not well suited to this type of environment. Therefore, I developed a unique wheel encoder system impervious to dirt and dust to solve this problem. This custom encoder system was built from special diametrically aligned magnets and Hall Switches to provide quadrature encoder output so that both the speed and direction of the wheels can be known. This unique system also allows for detecting slip of the wheels, an important consideration when this mower is operating on hills and inclines.

Traditional magnets are aligned on their axis, as shown in Figure 99(1). The magnets used for the ASMO wheel encoders are aligned diametrically, as shown in Figure 99(2). Diametrically aligned magnets mean the magnetic poles are sit next to each other rather than on top of each other—their sides are polarized as opposed to the top and bottom. Taking this approach allowed me to place a series of magnets around the wheel and pick up the North side of the magnet with one Hall Switch and pick up the South side of the magnet with a second Hall Switch placed directly next to the first.

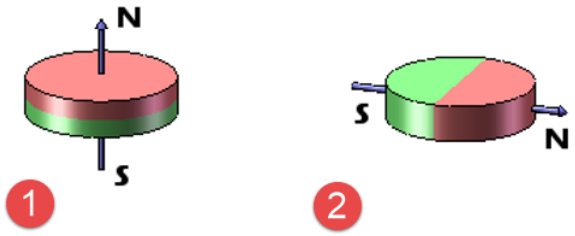


Figure 99. (1) Axially aligned magnets and (2) diametrically aligned magnets (K&J Magnetics, 2018)

The Melexis US5881 Hall Switch (Melexis, 2018) used for the both the Throttle Control system and the Laser Distancing system worked perfectly for this application as well. The Melexis Hall Switch has the ability to pick up the South Pole of a magnet on its face while able to pick up the North Pole of a magnet on its back. This meant I was able to place two of these sensors side-by-side, one facing up while the other faces down, to sense the single diametrically aligned magnet.

After inspecting the Bad Boy Outlaw XP wheel system, I decided to place the magnets on the backside of the brake rotors. The brake rotor attaches in between the mower's wheel and the hydraulic motor shaft and spins with both. A brake caliper is fixed to the hydraulic motor and connected to the parking brake. When the parking brake

is applied, the brake caliper tightens the brake shoes against the brake rotor, which prevents the wheel from turning. This is similar to how automotive brake rotors works. My thought was to place diametrically aligned magnets around the inside perimeter of the brake rotor and then place the Hall Switches inside a mount that could be afixed to the Hydraulic motor just like the brake calipers. I was able to source diametrically aligned magnets from K&J Magnetics for $0.50 each. The magnets were 1/4” diameter by 1/8” thick and have an N52 strength.

One issue with diametrically aligned magnets is that they tend to be the strongest on edge and not on their face. This meant I would need to have the Hall Switches fairly close to the magnet in order to pick them up. This is the reason I went with the stronger N52 magnets instead of the slightly less strong N42 magnets I had been using in the Throttle and Laser systems. The first step to mounting the magnets was to cut a 1/4" channel around the perimeter of the inside of the brake rotor (Figure 100).

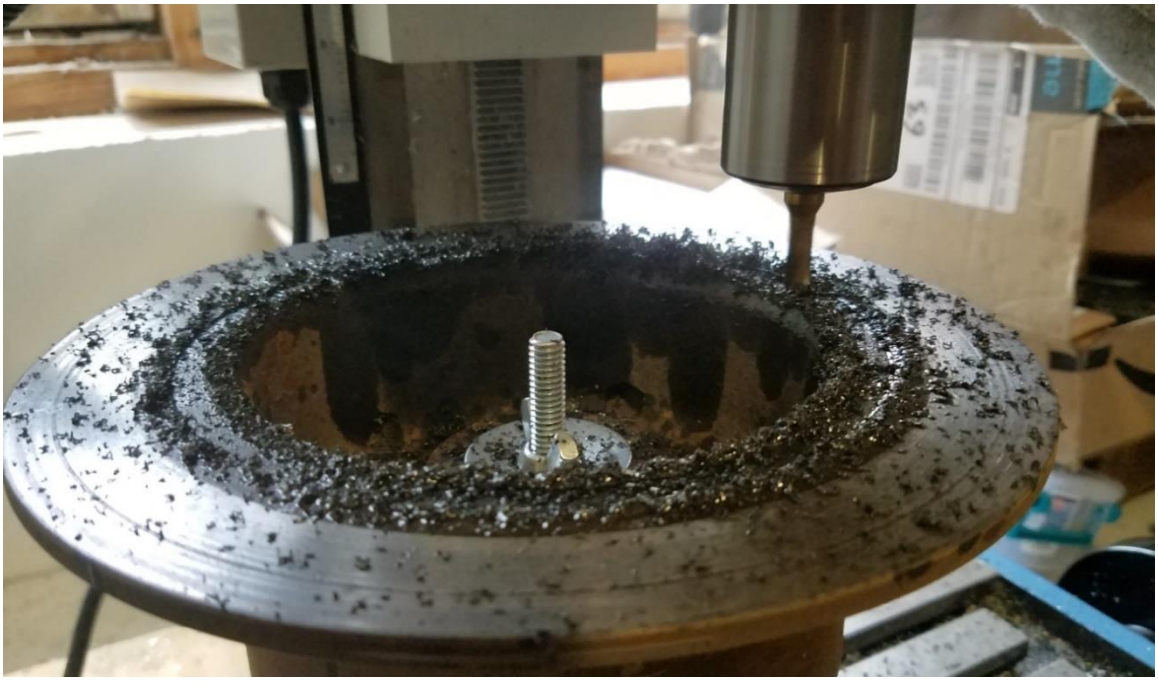


Figure 100. Cutting the channel on the rotor for the magnets

This channel helped with the alignment of the magnets and created a perimiter for me to work within. I cut the channel to be 5 3/8” in diameter. This meant if I spaced each 1/4” magnet 1/8” apart, I could fit 45 magnets inside the channel:

5.375” x π = 16.886” / (0.25” + 0.125”) = 45.029

Based on my testing, I needed some space inbetween each magnet. Otherwise, if they were too close together, they tended to create one large magnet. The 1/8” spacing between each magnet worked well. The next step was to create a 1/4” divot for each magnet, equally spaced inside the channel that was just created. I was able to precisely create each divot by incrementing the rotary table the brake rotor was mounted to by 8 degrees for each magnet (8 degrees * 45 magnets = 360 degrees).

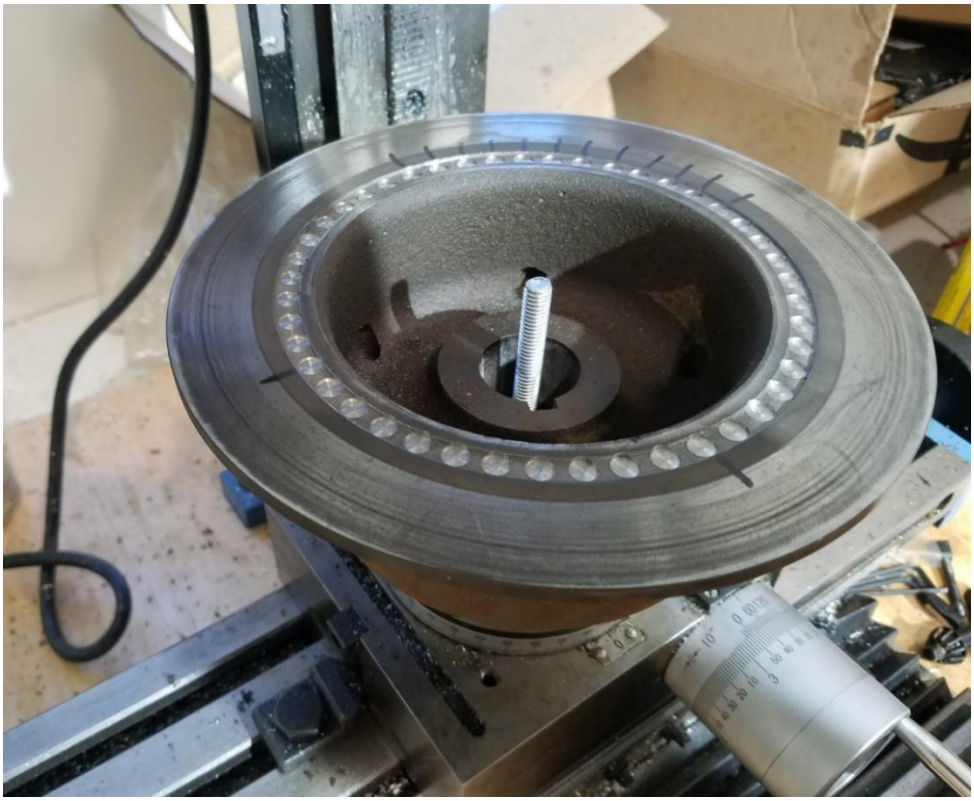


Figure 101. Slots milled on rotor and ready for magnets

It was interesting because I expected the magnets to simply stick in place. However, they tended to want to align towards the center of the rotor. I presumed this to be due to all the metal present in the very heavy rotor. In order to keep them in the correct alignment, with the North pole facing to the right, I had to use a dab of superglue. Once all magnets were in place, I put epoxy over them all to help hold them in place and protect them from the elements (Figure 102).

Figure 102. Rotor complete with magnets mounted and epoxied in place

Another interesting aspect regarding the placement of the magnets is that they were so strong that once there were a few of them in place, they no longer wanted to align towards the center of the rotor and instead wanted to align towards each other, which worked to my advantage. Once the epoxy dried, I test fitted the brake rotor back in place to ensure there was enough clearance between the magnets and the brake caliper (Figure 103). The fit was good and there was plenty of clearance.

Figure 103. Checking the magnet clearance of the brake rotor

Once both the left and right brake rotors were outfitted with magnets, I turned my attention to creating the mounting bracket for the Hall Switch sensors. I had previously decided to use the top motor mount bolts to hold a bracket. Doing so would allow the Hall Switches to be aligned with the top-center of the circumference of the magnet channel I created on the brake rotor. The bottom motor mount bolts were already being used to hold the brake caliper in place, so the top two bolts were long enough to hold a similarly shaped bracket (Figure 104).

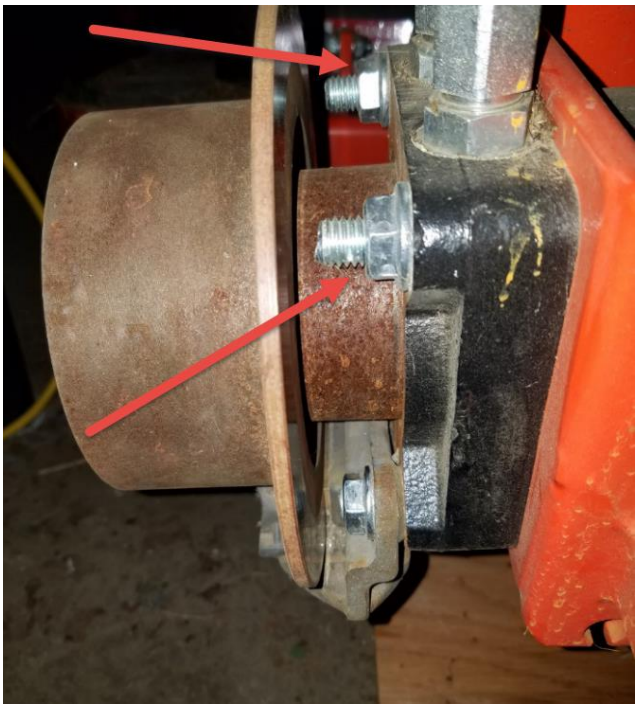


Figure 104. Proposed mounting location for the Hall Switch bracket

First, I designed the bracket to fit between the bolts. The inside of the bracket needed to be narrower than the rest of the bracket to accommodate the center of the Hydraulic motor. I decided to create the bracket from 1" thick aluminum bar stock (Figure 105).

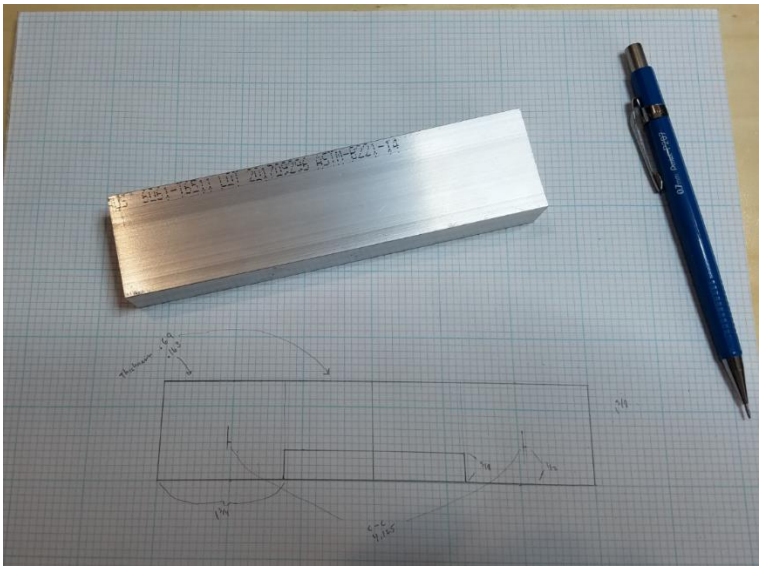


Figure 105. Hall Switch encoder bracket design

Once designed, I created a smaller version for alignment purposes along with two full size versions for the bracket itself. Figure 106 shows the brackets as they are nearing completion.

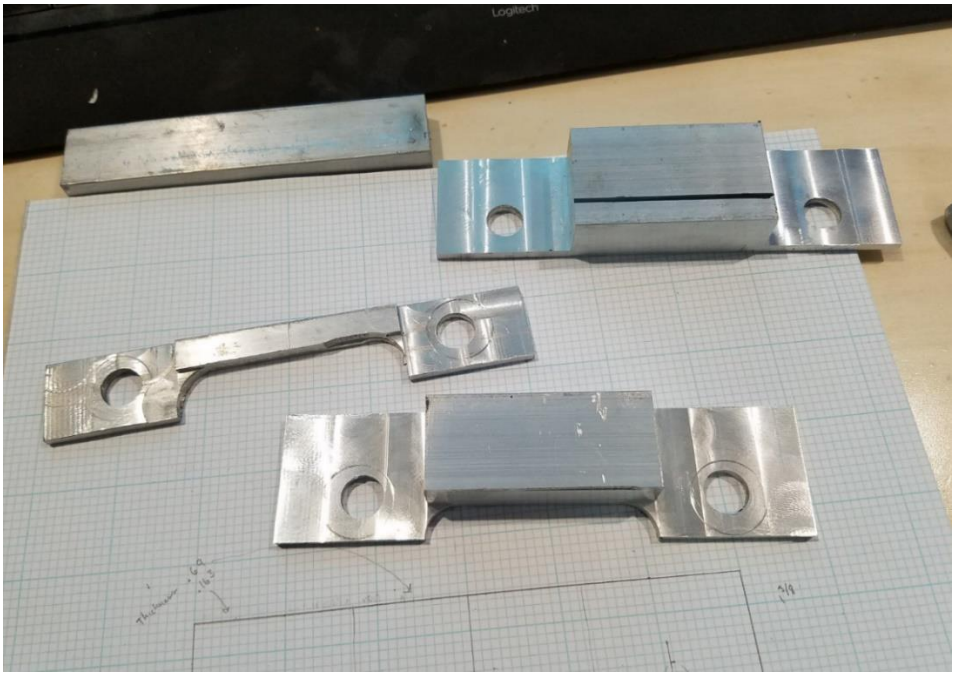


Figure 106. Partially completed Hall Switch encoder brackets

Due to the very tight required clearance between the Hall Switches and the magnets, the Hall Switches were recessed into the mounting bracket and epoxied into place.

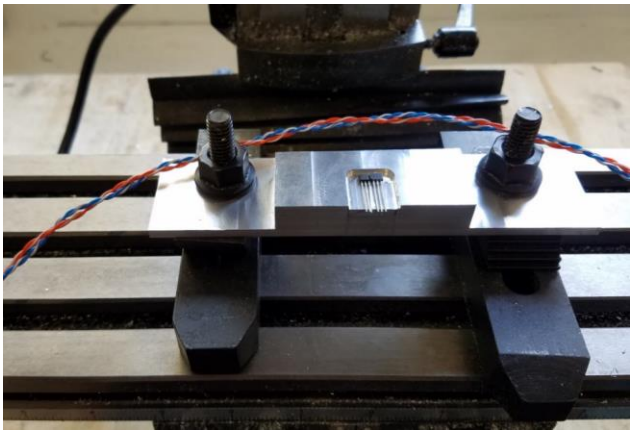


Figure 107. Milling out a channel to epoxy the Hall Switches

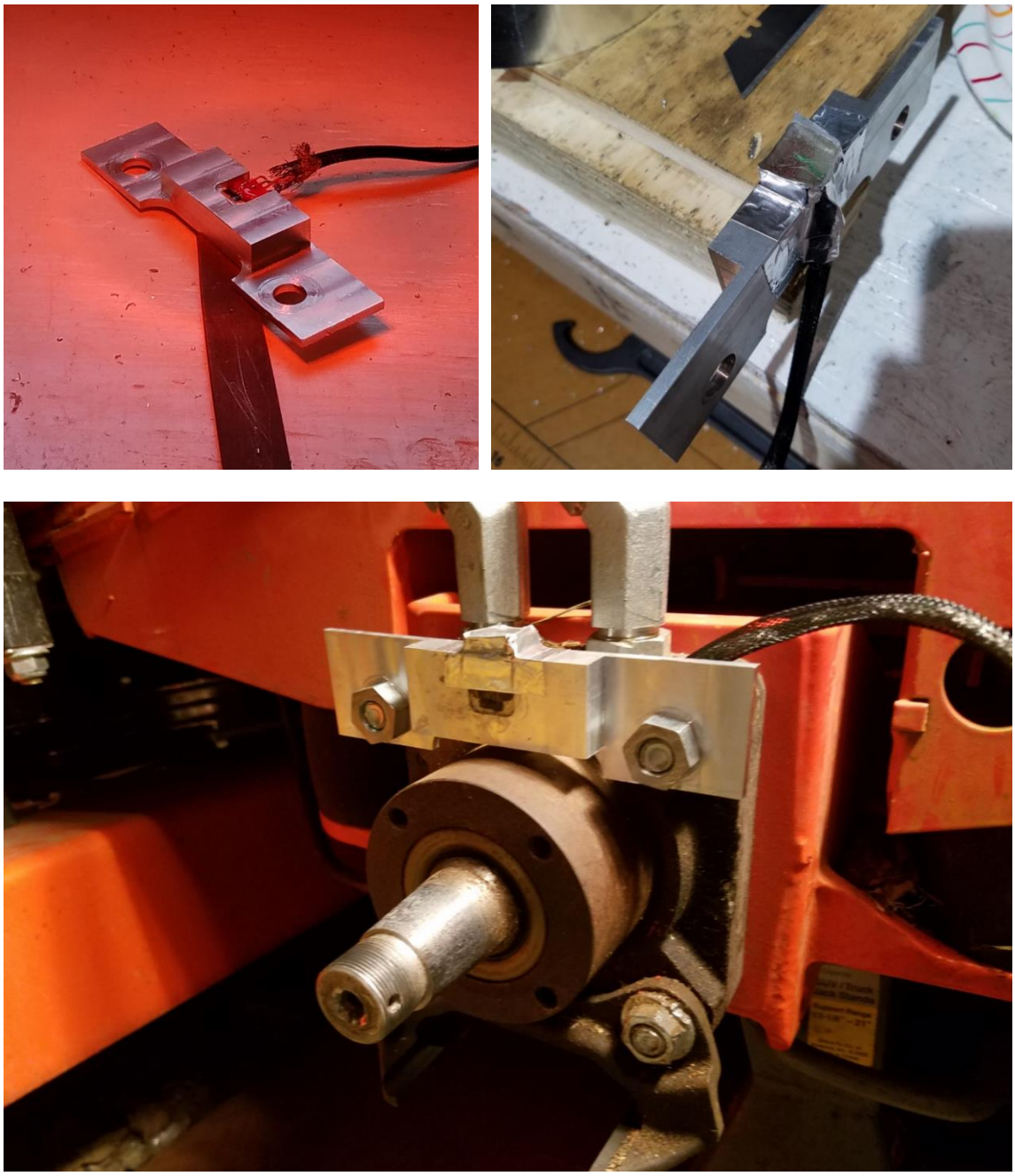


Figure 108. Hall Switch Bracket epoxy process and final mounting

The final mounting required some fine adjustments using spacers in between the back of the bracket and the motor mount. Using thin spacers allowed me to get the sensors as close as possible to the magnets without hitting them. The final clearance ended up being 1/16” of an inch. The sensors are low-voltage and operating in an area close to the

engine, which means there will be a lot of electrical noise. For this reason, I twisted the power and ground wires together and then twisted the two signal wires together. Finally, I twisted both sets of wires together and put the full length of wires inside wire shielding to help eliminate the possibility of signal degradation due to electrical noise from the mower.

Once the sensor brackets were mounted, I connected an oscilloscope to verify they were working as expected. Figure 109 shows a picture of the oscilloscope with the wheels running in a forward, clockwise, direction. The wheels were turning quickly. It is clear from the image that the sensors were working as expected. Channel 1 (in yellow) was leading channel 2 (in purple).

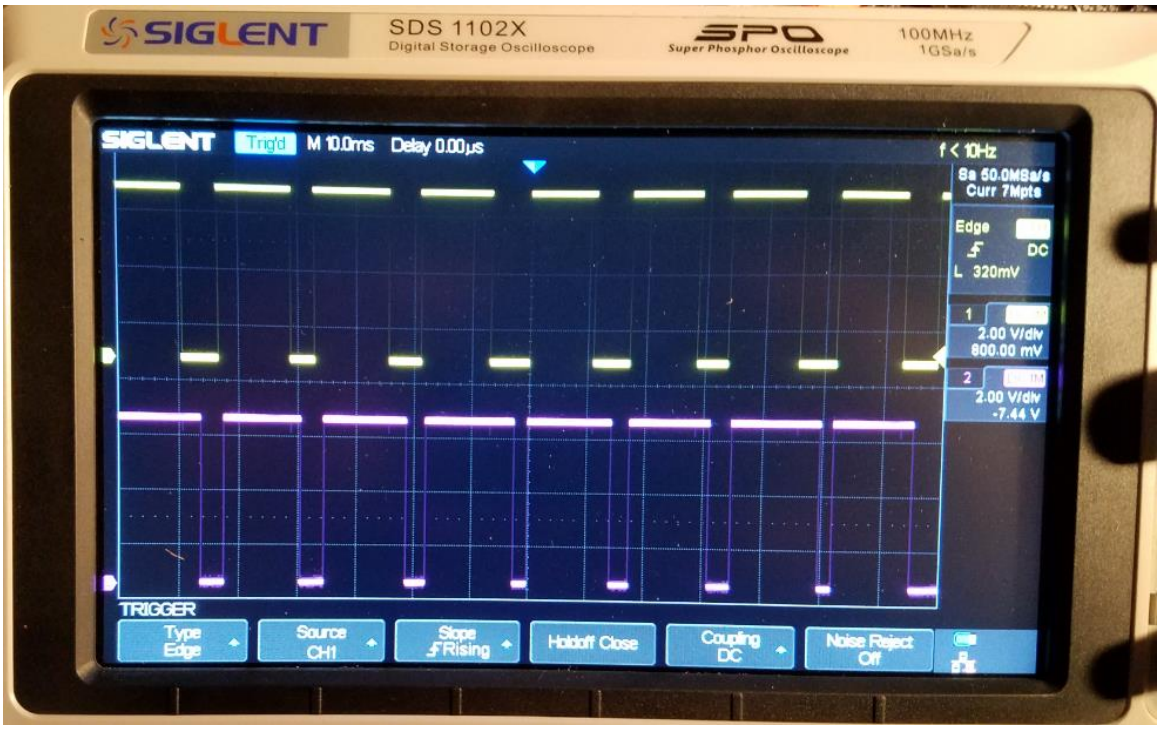


Figure 109. Wheels running forward, channel 1 (yellow) leading channel 2 (purple)

Figure 110 shows a picture of the oscilloscope with the wheels running backwards, in a counter-clockwise direction. In this case, the wheels were turning much slower, which is why the distance between edges is much greater than in Figure 109. You can see that in this case channel 2 (purple) is leading channel 1 (yellow). These results match the expected results from an optical encoder but instead were from a much more robust, magnetic, dirt-proof sensor.

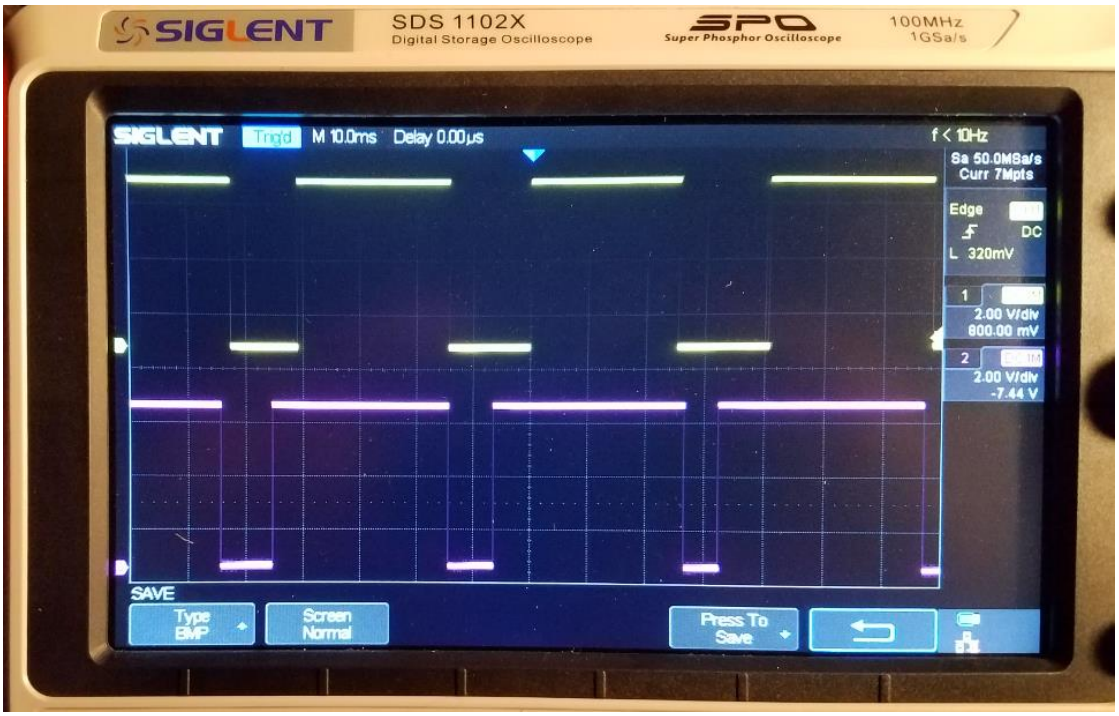


Figure 110. Wheels running backward, channel 2 (purple) leading channel 1(yellow)

The horizontal axis represents time while the vertical axis represents voltage. The voltage swing was about 5 VDC, which was expected given that the sensors are operating at 5 VDC. Figure 111 is a close-up of the trace shown in Figure 110. Each square represents 10ms. Given there are 45 magnets per revolution, assuming a consistent speed, the revolutions per second can be computed as follows:

38ms * 45 magnets per revolution = 1710ms per revolution (slow, Figure 111)

18ms * 45 magnets per revolution = 810ms per revolution (fast, Figure 112)

The above computations coincide with my visual observations of the wheels at the time, which was noted as approximately one revolution per second for fast and one revolution every two seconds for slow.

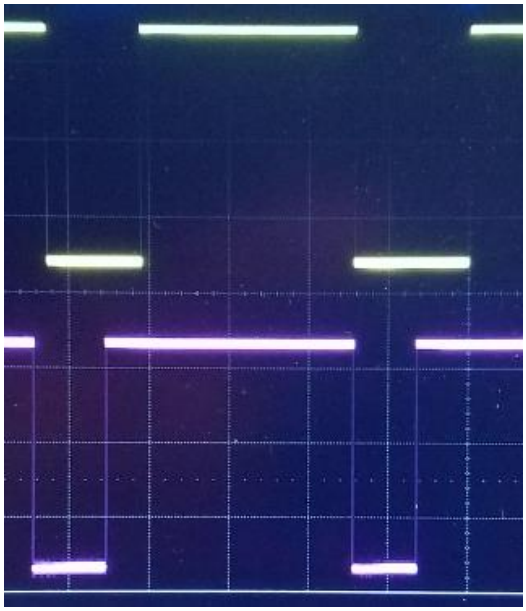


Figure 111. Close-up of the reverse encoder signal

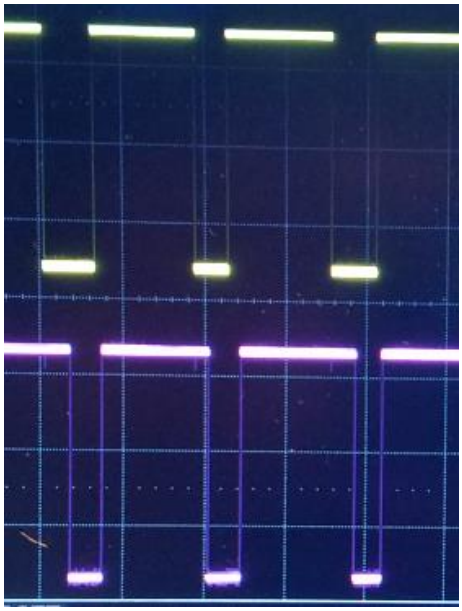


Figure 112. Close-up of the forward encoder signal

5.1.2 GPS

The GPS used in this project is the Adafruit Ultimate GPS breakout board (Figure 51). This GPS operates at 5 VDC, provides 10 Hz updates, and can track up to 22 satellites on 66 channels. This breakout board also can accept a coin cell battery to store ephemeris data for faster warm starts and to support a real-time-clock. One of the other nice features of this board is that it has an SMA connector for use with an external antenna. I made use of an external active antenna, which provided an additional 28dB of gain but also came with a 5-meter cable (Figure 95). This long cable allowed me to

mount the antenna higher on the mower, further away from the electromagnetic noise created by the spinning mower blades. The GPS outputs standard NMEA sentences over a serial connection and is directly connected to the MoteinoMEGA microcontroller.

Figure 113. Active GPS antenna connected to the GPS breakout board using an SMA to uFL adapter cable.

The GPS is used in the ASMO project to provide heading and general location information. The GPS is accurate to approximately 9 feet. As such, it is not accurate enough to be used for precise localization on its own. However, it does provide a secondary means of ensuring the mower does not operate outside a bounded area. This bounded GPS garden is discussed in the next chapter.

5.1.3 IMU

The Inertial Management Unit (IMU) used in this project is a breakout board from Adafruit (Figure 52), which is based on the 9-DOF Absolute Orientation Bosch BNO055

sensor. The BNO055 is a System in a Package (SiP) that integrates a 16-bit gyroscope, a triaxle 14-bit accelerometer, and a triaxle magnetometer together with a 32-bit cortex M0+ microcontroller. The microcontroller runs the Bosch Sensortec sensor software to provide fusion of the three sensor systems. The fused data is output as Quaternion, Euler angles, rotational vectors, linear acceleration, gravity and heading.

The sensor performs its own calibration based on the environment but does allow saving and reloading of the calibration data. This proved to be very useful as the rapidly spinning mower blades create an electromagnetic field that had a significant effect on the operation of this sensor. This was accounted for in the software interface. Additionally, to help reduce interference from the radio transceiver, the IMU was mounted in a plastic PCB enclosure on top of and away from the radio box (Figure 114).

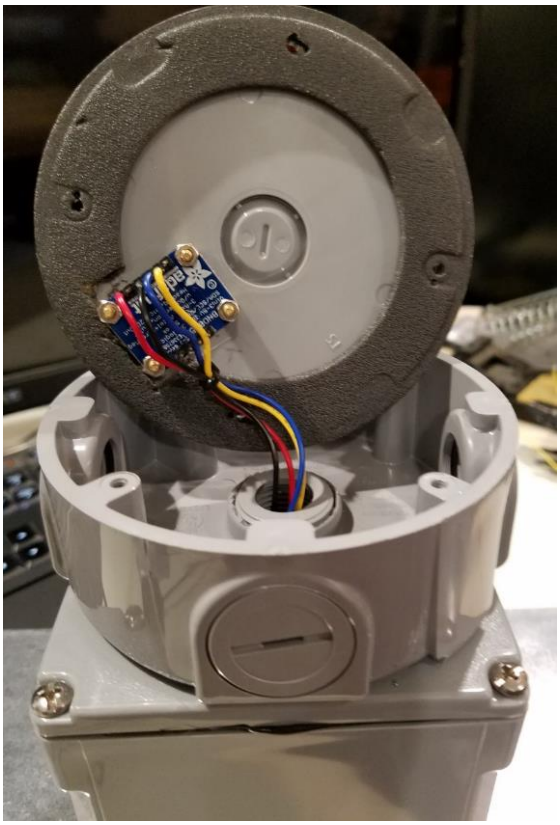


Figure 114. Showing the IMU mount

5.1.4 Microcontrollers

There were two microcontrollers used in the odometry system. The Microcontroller in the radio box is the MoteinoMEGA (Figure 43). It connects to the main computer through a USB serial connection. The MoteinoMEGA performs multiple duties. It has the radio transceiver for use with the remote control and interfaces to the relay board for controlling the parking brake and kill switch. These items were covered in Chapter 2. In addition to these functions, the MoteinoMEGA also interfaces to the IMU through I2C and the GPS through a UART. The MoteinoMEGA is based on the Atmega1284P, runs from 5 volts and operates at 16 MHz. This microcontroller is also compatible with the Arduino programming environment. The MoteinoMEGA has enough memory to run the ROS Serial library, which means it can directly expose its capabilities to the ROS environment.

The wheel encoders required hardware interrupts on the microcontroller to capture the tightly timed state transitions of the Hall Switches. While the MoteinoMEGA has a wide array of hardware inputs and outputs available, it only has three interrupt pins and one of them is used for the radio transceiver. This leaves only two interrupt pins, which is not enough to interface to both wheel encoders, which have two Hall Switches each. Therefore, a Teensy 3.2 microcontroller (Figure 42) was used to interface with the Hall Switches on the wheel encoders. The Teensy 3.2 microcontroller can have any of its input pins operate as a hardware interrupt. Additionally, while this microcontroller operates at 3.3 VDC, its digital input pins are 5 VDC tolerant. This means I did not need to create a level shifting circuit to connect the 5 VDC signal from the Hall Switches to the microcontroller. The Teensy 3.2 microcontroller is located in the main CPU box and

connects with the main computer through a USB serial connection. Each wheel encoder connects to the main CPU box through a screw-on 4-pin aviation connector (Figure 115). Two pins provide 5 VDC power and ground while the other two pins are connected to the Hall Switch sensor outputs.

Figure 115. Main CPU box showing (1) the wheel encoder Teensy and (2) the wheel encoder connectors

5.2 Software

There are several pieces of software that were written for the odometry system. For purposes of the odometry system, the software running on the MoteinoMEGA microcontroller interacts with the GPS and the IMU. As mentioned in Chapter 2, this microcontroller is also responsible for the parking brake, kill switch, and radio but those functions are not related to the odometry system and won't be covered again in this section. The software running on the Teensy microcontroller receives signals from the wheel encoders, turning the pulses into distance travelled measurements. The Teensy microcontroller also receives IMU data from the MoteinoMEGA, fusing it with the wheel encoder data to provide more accurate odometry data. Both the fused and unfused

odometry data are then exposed to the ROS environment for use with the higher level navigational systems.

5.2.1 Wheel Encoders

A Teensy microcontroller in the main CPU box is responsible for receiving pulses from the two-wheel encoder systems and turning these pulses into distance travelled. In order to perform this operation, the microcontroller needs to receive two sets of inputs from both the left and right wheel encoders. These two inputs are called Channel A and Channel B, which correspond to channels 1 and 2 from the previous hardware section. The Hall Switch producing pulses from the North pole of the magnet is Channel A while the Hall Switch producing pulses from the South pole of the magnet is Channel B.

As was shown in Figure 109, when the mower is moving forward, the wheels turn clockwise, and Channel A leads Channel B. Similarly, Figure 110 shows that when the mower is moving in reverse, the wheels turn counter-clockwise and Channel B leads Channel A. There are 45 pulses for each polarity for each complete rotation of the wheel. Therefore, 45 Channel A leading pulses equates to one forward rotation while 45 Channel B leading pulses equates to one reverse rotation. Counting the number of pulses for each channel from both wheels allows the system to compute the distance travelled based on the circumference of the wheels.

Before any distance information can be computed, the pulses need to be accumulated in such a way that the previous state is taken into account. As pointed out by Daniel Gonzalez on his Mechanical Engineering Blog (Gonzalez, 2012), evaluating the encoder state of both channels on the rising and falling edge of each signal allows for a four-fold increase in the quadrature encoder precision. There are 45 magnets in the

wheel encoder. Each magnet has a positive and negative polarity, which increases the pulse count to 90 for the rising edge. Since I am also tracking the falling edge of both polarities, this increases the number of pulses to 180 per revolution of the wheel. This means that there will be four times the number of interrupts and four times the state to be tracked. However, the Teensy 3.2 microcontroller runs at 72 MHz, which is substantially faster than the average 16 MHz Arduino compatible board. I experienced no issues accumulating the state.

In order to know whether the overall tick count needed to be incremented or decremented whenever an interrupt occurs, I decided to create a table to represent the various possible states (Table 11). This table tracks the possibility of the Channel A and Channel B being in four possible states, given that the interrupt is being trigged on a change, not just a rising or falling edge. The output can be accumulated over time.

Table 12. Encoder state accumulator output

Channel A Previous	Channel B Previous	Channel A Current	Channel B Current	Output
True	True	False	True	-1
True	True	True	False	1
False	True	False	False	-1
False	True	True	True	1
False	False	True	False	-1
False	False	False	True	1
True	False	True	True	-1
True	False	False	False	1

Tracking the previous state of channels A and B allows the system to know whether or not the new tick from the encoder should add or subtract to the total number of ticks accumulated. Once the information was put into a table, it was a simple matter of translating the table into a function. The ParseWheelEncoder function shown below takes a wheelId (LEFT or RIGHT) and checks the previous A and B states against the current states. This function is called as part of the interrupt handler for each of the four inputs.

```
int ParseWheelEncoder(int wheelId){
  if(_channelAPrev[wheelId] && _channelBPrev[wheelId]) {

    if(!_channelASet[wheelId] && _channelBSet[wheelId]) return -1;
    if(_channelASet[wheelId] && !_channelBSet[wheelId]) return 1;

  }
  else if(!_channelAPrev[wheelId] && _channelBPrev[wheelId]) {

    if(!_channelASet[wheelId] && !_channelBSet[wheelId]) return -1;
    if(_channelASet[wheelId] && _channelBSet[wheelId]) return 1;

  }
  else if(!_channelAPrev[wheelId] && !_channelBPrev[wheelId]) {

    if(_channelASet[wheelId] && !_channelBSet[wheelId]) return -1;
    if(!_channelASet[wheelId] && _channelBSet[wheelId]) return 1;

  }
  else if(_channelAPrev[wheelId] && !_channelBPrev[wheelId]) {

    if(_channelASet[wheelId] && _channelBSet[wheelId]) return -1;
    if(!_channelASet[wheelId] && !_channelBSet[wheelId]) return 1;

  }
}
```

Once the ticks were being tracked, the next step was to turn this data into more useful distance travelled information. The distance travelled is a function of both the left and right wheels, which can be turning at slightly different amounts even when the mower is traveling straight due to wheel slip. The overall distance travelled is the result of the difference between the left and right wheels. Since the mower uses differential

drive in order to turn, taking the difference between the left and right wheel encoders into account allows for the direction of travel to be known. This means that when the mower is turning right, there will be more pulses coming from the left wheel than from the right wheel. Likewise, when the mower is turning left, there will be more pulses coming from the right wheel than from the left wheel. Tracking the relationship between the left wheel pulses and the right wheel pulses allows the system to know if the mower is turning left or turning right.

The distance for the left and right wheels can be computed as follows:

```
double dist_left_mm = dist_left_encoder_counts / LEFT_ENCODER_COUNTS_PER_MM;
double dist_right_mm = dist_right_encoder_counts / RIGHT_ENCODER_COUNTS_PER_MM;
```

Once the left and right distance is known, the total distance travelled can be computed:

```
double right_left_m = (dist_right_mm - dist_left_mm) * MM_TO_M;
double theta = right_left_m / WHEELBASE_M;

double distance_m =  (dist_right_mm + dist_left_mm) * 0.5 * MM_TO_M ;
```

Using the above two quantities, the position in two a dimensional Cartesian plane can be computed using basic trigonometry:

```
double vx = distance_m * cos(theta);
double vy = distance_m * sin(theta);
```

Additionally, since theta is in radians, a heading in degrees can be calculated by multiplying theta by (180.0/π).

5.2.2 IMU

As with all of the Adafruit breakout boards, they made a library available for interacting with the sensor. This library exposed a set of functions that provided access

to all the data necessary from the BNO055 sensor required to populate a ROS IMU message structure, as shown in Table 13 (ROS, 2018b).

Table 13. ROS IMU message structure (ROS, 2018b)

Name	Type	Description
header	Header	The header includes a sequence, a timestamp that should be the acquisition time from the IMU
orientation	Quaternion	This is a structure representing an orientation in free space in quaternion form: float64 x, y, z, w
orientation_covariance	float64[9]	The covariance matrix is set to -1 as this information is not available from the BNO055.
angular_velocity	Vector3	This is a structure representing a vector in free space. Rotational velocity is in rad/sec.
angular_acceleration_covariance	float64[9]	The covariance matrix is set to -1 as this information is not available from the BNO055.
linear_acceleration	Vector3	This is a structure representing a vector in free space. Acceleration is in m/s^2.
linear_acceleration_covariance	float64[9]	The covariance matrix is set to -1 as this information is not available from the BNO055.

The orientation, angular velocity, and linear acceleration are all available from the BNO055 at a rate of 100 Hz. The IMU information is published at a slower rate of 10 Hz inside the ROS environment. The angular velocity is obtained from the gyroscope. The linear acceleration is obtained from the accelerometer. The X, Y, and Z orientation

values are obtained from the fused absolute orientation values computed by the CPU onboard the BNO055 itself.

The BNO055 sensor is an absolute orientation sensor. This means that once the device is calibrated, it is able to provide absolute heading information with respect to the Earth's magnetic North Pole. In order to obtain absolute orientation, the device needs to be calibrated. Calibration of the device requires moving the sensor through a series of motions such as a figure eight to calibrate the accelerometer and moving the device around a fixed axis 90 degrees at a time through a full 360-degree rotation. Once calibrated, the device retains the calibration settings until switched off. The interface library does provide a means of obtaining and restoring the calibration data, so it wouldn't have to be recalibrated every power cycle.

The issue with calibration is that it is very sensitive to nearby electromagnetic fields. I ran into issues with the high-speed spinning mower blades creating enough of a magnetic field to disturb the accuracy of the device when calibrated without the mower blades running. I was able to save calibration data from when the mower desk was both on and off, which improved the situation. In the end, I decided to forgo the use of absolute orientation values and rely on the relative orientation values that were always available and always accurate. The main data point I made use of from the IMU was the X orientation value. In an uncalibrated state, the X orientation contains the yaw of the device relative to its location when first turned on. As will be shown in the next section, the change in this relative value was sufficient to perform sensor fusion with the wheel encoders to improve the accuracy of the odometry system.

5.2.3 Sensor Fusion

The idea of sensor fusion is to combine multiple sensor values together in order to produce a result that is more accurate than either individual sensor. The wheel encoder system used in the ASMO project is subject to accumulated errors due to wheel slip. If one wheel turns more than another wheel, not because the mower is actually performing a turn, but because the wheel slips on the grass, then according to the wheel encoder counts, the slipping wheel is causing a larger change in distance travelled than actually exists. This can result in the mower appearing to make turns when it is actually going straight. This error shows up in the theta variable discussed in section 5.2.1, which is the difference between the right and left wheel travel divided by the wheel base.

The wheel slip that occurs when the mower is going straight is minimal when compared to the wheel slip that occurs when the mower is performing a turn. The reason for this is that zero-turn mowers, such as the Bad Boy Outlaw XP used in this project, rely on skid steering to perform turns. A skid steer drive, also known as a differential drive, utilizes the idea of wheel slip to perform turns. This is unlike the steering mechanism in more traditional mowers, which use rack and pinion steering to angle the front wheels to make turns. Rack and pinion steering results in a wider turn radius. A differential drive can make a zero-turn or "spin-in-place," which inherently means the wheels are slipping as the turn is performed—as noted by Wu, Xu, and Wang, "when the [differential steering] vehicle turns, the longitudinal slip of the wheels on both sides will be different" (Wu, 2013). These mowers are popular in landscaping because their inherent agility allows for tighter mowing around objects.

In order to determine to what extent wheel slip would pose a problem for the encoder system used in the ASMO project, I devised a test. The test was to drive the mower around a square path while outputting ROS odometry messages. The odometry messages would be displayed in the ROS RVIZ program. Ideally, if wheel slip was not an issue, the odometry path travelled would take the form of a closed box once the test program completed and the mower returned to its starting position. This seemingly simple test was more difficult to implement than expected. The first problem I ran into was writing the code to get the mower to produce a 90 degree turn. I was not yet tied into the ROS navigation system, so I couldn't rely on that system to make a 90 degree turn. I attempted to make a time-based turn (e.g. turn for 500ms) but this did not result in a 90 degree turn 100% of the time. Sometimes the turn would undershoot 90 degrees while other times it would overshoot 90 degrees, even though the amount of turn time was consistent. The reason for the inconsistency could sometimes be attributed to wheel slip and other times due to other mechanical reasons—such as the drive actuator taking slightly more time to start or stop. There were simply too many variables to try and control the system in an open-loop manner with no feedback.

Instead of using a time-based approach to make a 90-degree turn, I decided to make use of the IMU X orientation value, otherwise known as the heading. This heading value was accurate and consistent, at least from a relative starting position. It was also available as a floating-point value from 0 (inclusive) to 360 (exclusive). I modified my test program to capture the current heading value just before the turn started. I then added 90 to this value to calculate a target value. If this resulted in a target value that was greater than 360, I subtracted 360 from the target value. This was done to account for

wrapping around the 360-degree mark. The turn was then started and allowed to continue until the current heading value was greater than the target heading value. One additional item I had to account for was to ensure this check didn't occur until after the current heading value was greater than 0. This last check was necessary because if the target heading value was less than 90, it meant that the current heading value was between 270 and 360 when the target heading was computed. This meant that the current heading value was going to initially be greater than the target heading value, until the current heading value passed 360 and wrapped around to the 0-degree mark.

Once the above changes were made to the test program, the mower was much closer to making a perfect 90 degree turn. The last variable that had to be taken into account was the amount of time required to actually stop moving. This value varied based on a number of conditions, such as the ground type, mower weight, and mower speed. The ground type was a factor based on whether the mower was operating on grass or gravel, which were the two most common ground types used during my testing. Table 10 outlines the coefficient of friction values for different ground types. The weight of the mower varied based on the presence of an operator. The speed of the mower varied based on how fast I programmed the mower to move. Once these variables had been accounted for, I was able to perform a successful 90 degree turn. These variables were accounted for by initiating the stop of turn process before 90 degrees was reached. This allowed the mower time to stop moving, once the actuators were commanded to stop, allowing the mower to settle at 90 degrees.

The test program was now able to drive the mower in a square path. The test program drove the mower forward for 15 seconds, turned 90 degrees, then drove forward

15 seconds, turned 90 degrees, then drove forward another 15 seconds before making the final 90-degree turn and driving the last 15 seconds to close the square. Time was used instead of distance for this test as distance was being measured. The mower ended up within 12" of where it started. The RVIZ program displayed the results of the wheel encoders as captured from the odometry messages (Figure 116). The displayed output was very far from what was observed. The odom mark in Figure 116 represents the starting position of the mower while the base_link mark indicates the ending position of the mower. Each grid cell in the figure represents 10 feet. It is clear from the Figure that these two locations were significantly different from the measured 12" difference.

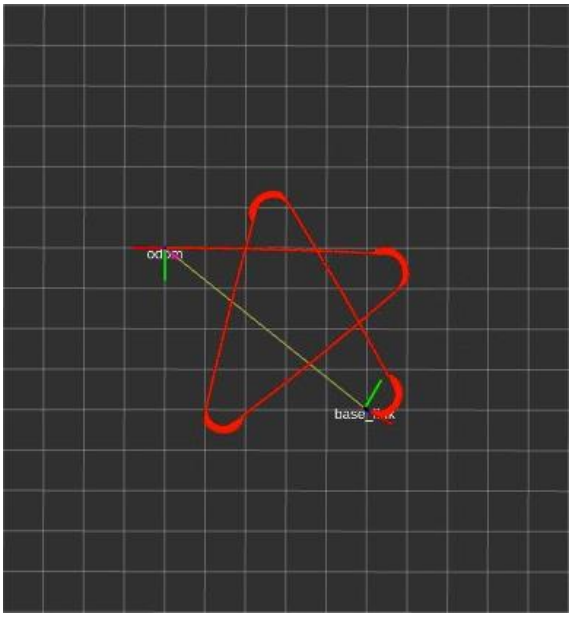


Figure 116. Odometry path from raw wheel encoder data

The next step was to fuse the IMU data with the encoder data in order to provide more accurate wheel odometry. There are various approaches available for fusing data together. ROS provides a robot_localization package for fusing sensor data together. This package provides two filters for fusing data—the Extended Kalman Filter (EKF) and

the Unscented Kalman Filter (UKF). A Kalman Filter is a Bayes filter that uses a Gaussian distribution to estimate probability distributions. The Kalman Filter is a good state estimator for linear systems. Gaussian distributions break down when used with nonlinear systems. Since most systems in robotics are non-linear, the Kalman filter alone is insufficient. The Extended Kalman Filter is an extension of the Kalman Filter that operates by transforming nonlinear data into linear data at discrete time points. The Unscented Kalman Filter is a variation of the Extended Kalman Filter and can work better in some circumstances.

In addition to investigating the robot_localization package, I also investigated another library more suited to embedded microcontrollers called TinyEKF (Levy, 2017). The TinyEKF library was implemented by Simon D. Levy, professor of Computer Science at Washington and Lee University. It is a C/C++ implementation of the Extended Kalman Filter for use in constrained environments such as those found with Arduino microcontrollers. I performed testing with the TinyEKF library but had difficulty tuning the covariance matrix values in order to obtain workable results. This situation is not uncommon as proper operation of the Kalman filter requires knowledge of sensor noise statistics, which is not generally known (Åkesson, 2007). Rather than spending additional time trying to tune the covariance values to get the EKF working properly, I decided to revisit the original problem.

Based on my observations, the principal problem with the wheel encoders occurred during turning operations. As discussed in section 5.2.2, the IMU has an orientation X value that contains the relative or absolute heading in degrees. The heading is absolute once the device is calibrated and relative until that time. Since I am only

considering the difference between the previous heading and the current heading, it doesn't matter if the value is relative or absolute. Not requiring this value to be absolute with respect to the Earth's magnetic North Pole simplified use of the sensor. This heading value is what I used to perform the 90-degree turn for the first test. I decided to replace the theta value computed from the wheel encoder distances and replace it with a theta value computed from the previous and current value of the IMU heading. I took the difference between the previous IMU heading and the current IMU heading, which was in degrees, and multiplied it by $\pi/180$ in order to get radians. I then used this value in place of the value computed solely from the wheel encoders. Once I performed this change, I re-ran my original 15 second drive test. The results were much improved over the use of the wheel encoders alone (Figure 117).

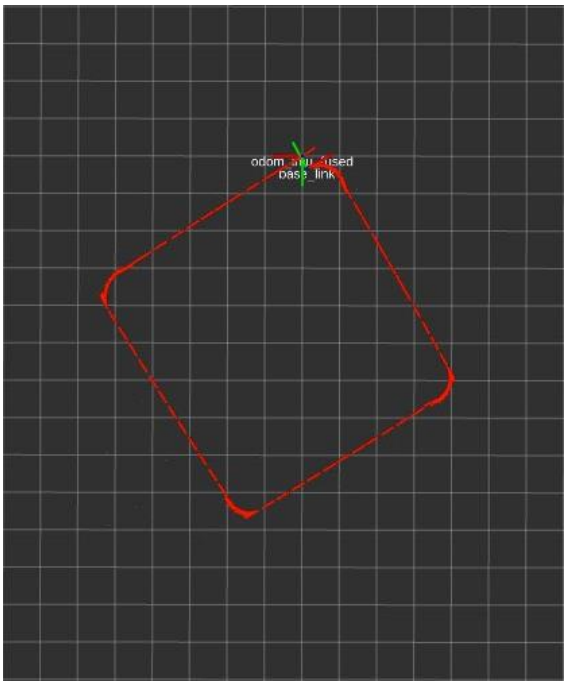


Figure 117. Odometry path after fusing the raw wheel encoder values with the IMU yaw

The measured distance of the starting and ending position of the mower was within 12". This was also the case with the odometry data as visualized with the ROS RVIZ program.

5.3 Summary

This chapter covered the unique custom Wheel Encoder system built for the ASMO project. Included were details of the diametrically aligned magnets used on the mower brake rotors that provided input to the dual Hall Switch sensors. It was then shown how these Hall Switch sensors were used to output quadrature encoder signals from which both the speed and the direction of travel could be inferred. The Inertial Management Unit used in this project was also detailed. An experiment was outlined demonstrating the problems computing odometry information from wheel encoder data alone. Various methods of combining the IMU data with the wheel encoders was then discussed. In the end, the simplest approach to increasing the accuracy of the odometry data was to replace the heading calculated from the encoder data with the heading delta computed from the output from the IMU. Lastly, the experiment with the fused data was performed and the successful results shown.

Chapter 6

Operator Interface and Map Making

ASMO is designed to ultimately operate autonomously. However, ASMO does require operator input in order to define a work area. The system was designed in such a way as to allow an operator to control ASMO either through a remote-control system (Figure 45) or through traditional mower controls (Figure 55). To facilitate the traditional approach to mower control, the operator seat and control arms were left intact. Doing so provided a more comfortable and familiar interaction with the mower than could be had through the remote-control interface. The remote-control interface was convenient for some operations but proved difficult to master. The joystick present on the remote was also not accurate enough for precision driving operations such as parking inside a cluttered garage. For general purpose operations, use of the control arms worked best. In order to retain use of the control arms, their mechanical linkages were replaced with electronic ones to provide the drive-by-wire functionality required for autonomous operation. Other operator interfaces were also modified to facilitate computer control. These modifications are covered in this chapter.

ASMO is a complex system composed of many software and hardware subsystems. Several different debugging tools from the ROS environment proved useful during the development process. These tools are outlined in this chapter along with how these tools were used to develop a work area map. The work area map is created during the training process and defines the area the mower is able to operate within. Creation of the work area map is currently a semi-automated process requiring the operator to drive the mower around the work area in training mode while the ASMO system accumulates

laser and odometry data. This data is then manually converted to a navigational map using the ROS map making tools covered in this chapter.

6.1 Control Panel

The original Bad Boy Outlaw XP control panel is shown in Figure 55. The original control panel consisted of mechanical switches. These switches drove higher power relays. The relays were used to perform operations such as moving the mower deck up and down, turning the mower deck blades on and off, as well as starting and stopping the engine. Additional controls were present for the engine throttle and choke. The original control panel also had a tachometer to display engine speed. In order to bring the mower under computer control, but still provide an interface to the operator, these mechanical controls had to be replaced with electronic ones. The computer would ultimately perform the operation, such as turning the mower deck on or off, but the operation could be initiated by the operator or by the computer. The use of physical controls near the operator provided a more natural manual interface than use of the remote control alone.

6.1.1 Hardware

An important aspect to choosing the type of electronic controls to use was that they did not allow for the state of the mower to become out of sync with the underlying function. For example, I had considered using a linear potentiometer to allow the operator to control the throttle. This would be more natural to the operator as the throttle could be increased or decreased simply by sliding the potentiometer up or down. The problem with this approach is that the operator could leave the potentiometer in one

position, but then the computer might adjust the throttle to another position. Now the state of the potentiometer did not match with the actual state of the engine throttle. This could lead to unanticipated behavior when the system started back up at some future point. Instead of using a linear potentiometer, I chose to use momentary push buttons—one for increasing the throttle and one for decreasing the throttle. Taking this approach never allowed for the physical state of the controls to become out-of-sync with the actual state of the system.

An additional consideration had to do with safety, which was always at the forefront of any decision in this project. I did not want important operations such as turning the mower deck on or enabling autonomous mode to be an operation that could accidentally occur. The original mower blade switch provides a good example of how to avoid turning the mower deck on accidentally. The original mower blade switch required the operator to pull-up on the switch in order to enable the cutting blades. You could accidentally lean on the switch, thereby pushing it down, to turn it off but it would take a lot more work to accidentally turn them on by pulling up on the switch. In choosing the setup of the new control panel, I made sure to think through these types of accidental situations.

In order to prevent the accidental operation of a function, I chose to implement a two-step approach to the major functions. The first step involves a toggle switch. The toggle switch has a cover that must be flipped-up, exposing the switch itself. Once the cover is up and the toggle switch exposed, the toggle switch must be flipped to the on position. Once this occurs, there is a visual cue to the operator in the form of a momentary push-button switch that lights up the same color as the rocker switch. The

second step in the two-step process requires the momentary push-button to be pressed in order to activate the associated function. Once the function is active, it can be deactivated by either pressing the momentary switch again or by closing the rocker switch cover, which also causes the rocker toggle switch to turn off. Accidentally leaning on a switch caused the cover to close, thereby disabling the function. You could not accidentally press a button to enable a function without first enabling the rocker switch. This multistep process helped to prevent accidental operation of dangerous functions. The control panel is shown in Figure 118 with associated controls in Table 13.

Figure 118. Custom mower control panel

Table 14. Custom mower control panel functions

Picture ID	Control Description
1	Throttle up button.
2	Throttle down button.
3	Top rocker to enable engine start / bottom button to start the engine. Once the engine is running, either switch can turn it off.

4	Top rocker to enable mower deck start / bottom button to start the mower deck. Once the mower deck is running, either switch can turn it off.
5	Top rocker to enable autonomous/training mode. The bottom button starts training mode. Once autonomous/training mode is enabled, either switch can disable (stop) it.
6	The original oil light.
7	The original hour meter / tach
8	An 2x16 LCD display (daylight readable) for outputting debugging information

The various switches for the control panel feed into digital input lines of a Teensy 3.2 microcontroller mounted under the control panel. This microcontroller also interfaces to the 2x16 LCD display through digital output lines. The microcontroller connects to the main computer through a USB Serial connection.

6.1.2 Software

An advantage to using the Teensy 3.2 microcontroller for the control panel interface is that this microcontroller has enough resources to run the ROS Serial library. As mentioned in previous sections, the ROS Serial library allows program functions on the microcontroller to be directly exposed for use in the ROS environment. As such, the microcontroller running the control panel interface does not control any of the associated functions directly. Instead, it simply sends ROS messages indicating the requested change of system state. There are four different messages that can be emitted from the control panel software (Table 14).

Table 15. Control Panel messages

Message Name	Parameter Description	Parameter Value
ThrottleButton	Down/Up	False = Down, True = Up
IgnitionButton	Stop/Start	False = Stop, True = Start
MowerDeckButton	Off/On	False = Off, True = On
AutonomousModeButton	Off/On	False = Off, True = On

In addition to emitting the state of the control panel in the form of ROS messages, the control panel also provides a single function to update the 2x16 LCD display called UpdateDisplay. The UpdateDisplay function takes two string parameters. The first controls the data displayed on line 1 of the display while the second controls the data displayed on line 2 of the display. The main computer uses this function to indicate the state of relevant information, such as the engine speed and throttle position. This function was also useful for displaying debugging information such as the wheel encoder ticks or Hall Switch counts from the engine tachometer.

6.2 Control Arms

The control arms are shown in Figure 29. As discussed in Chapter 2, the control arms were disconnected from the hydraulic motors and connected to linear actuators in order to bring the drive wheels under computer control. However, this change did not require removal of the control arms themselves. As the mower used in the ASMO project was also the mower I used to maintain my yard, the ability to easily disconnect the mechanical linkages from the linear actuators and reconnect them to the control arms proved very useful throughout the summer-long development process. Additionally, my

original plan to utilize a joystick to drive the mower around for training purposes was problematic given the imprecise control the joystick provided over the system. Therefore, I decided to create a microcontroller interface to the control arms, so they could be used in concert with the linear actuators in order to drive the mower around. This made manual control of the mower much more natural.

6.2.1 Hardware

As shown in Figure 29, the control arms are not directly connected to the hydraulic drive motors. They are connected to the drive motors indirectly through a series of mechanical linkages. These linkages include a shock absorber and spring system that allow the tension in the arms to be adjusted. The control arms attach to the input of the spring and shock absorber system and the output of this system is the control rod connecting directly to the hydraulic motor control. This indirect connection is what allowed me to disconnect the control arm from the control rod without affecting the operation of the control arm itself. The control arms stayed in place and continued to have the same feel as they had when connected to the hydraulic motors. This is what allowed me to reuse the control arms for remote control of the drive wheels. I just required a way to read the position of the control arms with a microcontroller.

The original function of the control arms was to rotate a small hydraulic control valve thereby controlling the speed and direction of a hydraulic motor. My initial thought was to simply have the control arms rotate a potentiometer instead of the control valve. I could then read the potentiometer value using an analog input on a microcontroller. This would be very similar to the approach used for the linear actuators when controlling the drive motors. However, I did not believe I could protect a

potentiometer from the harsh mowing environment. Therefore, I looked for alternatives. I found the MAE3 Absolute Magnetic Kit Encoder manufactured by US Digital (Figure 119).

Figure 119. MAE3 Absolute Magnetic Kit Encoder from US Digital

The Absolute Encoder makes use of a magnet and Hall Effect sensor to accurately output an analog voltage in much the same way as a potentiometer. A magnet gets attached to the end of a shaft whose rotational position is to be measured. In order for this to work, I recreated a pivot arm assembly that operated in a similar way to the hydraulic control valve I was replacing with the sensor. Figure 120 shows the creation of the pivot bracket along with the bronze bushings that allowed the new sensor arm to pivot with the control arm.

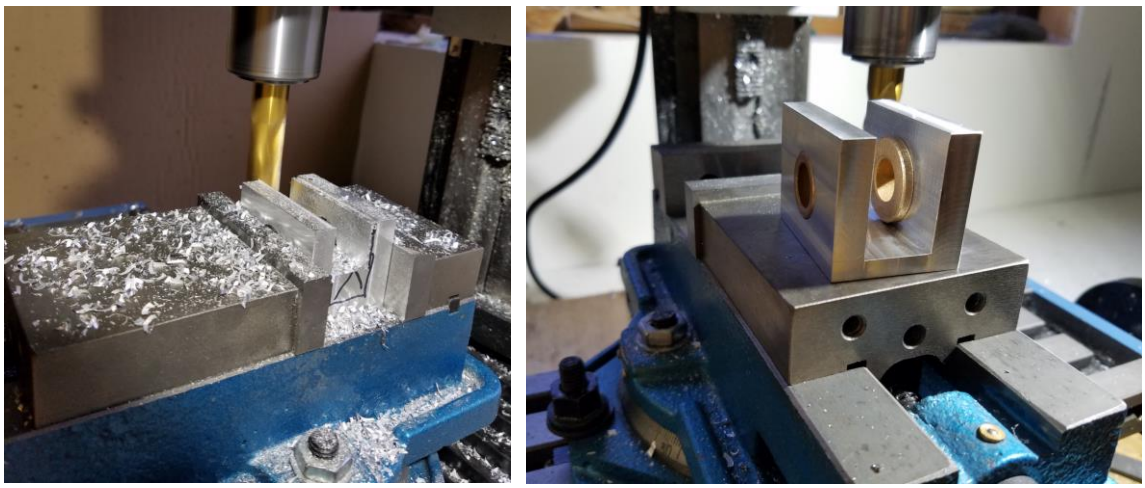


Figure 120. Creating the control arm sensor bracket

Once the pivot bracket was created, I turned down a raw aluminum rod to a ½ inch diameter, so it would fit through the bushings and the new pivot arm. The end of this rod was turned down further to a ¼ inch to fit inside the magnetic encoder sensor. This also allowed the magnet that came with the sensor to be mounted. The completed assembly, with the newly created control arm, and with the magnet mounted on the end of the rod, can be seen in Figure 121.

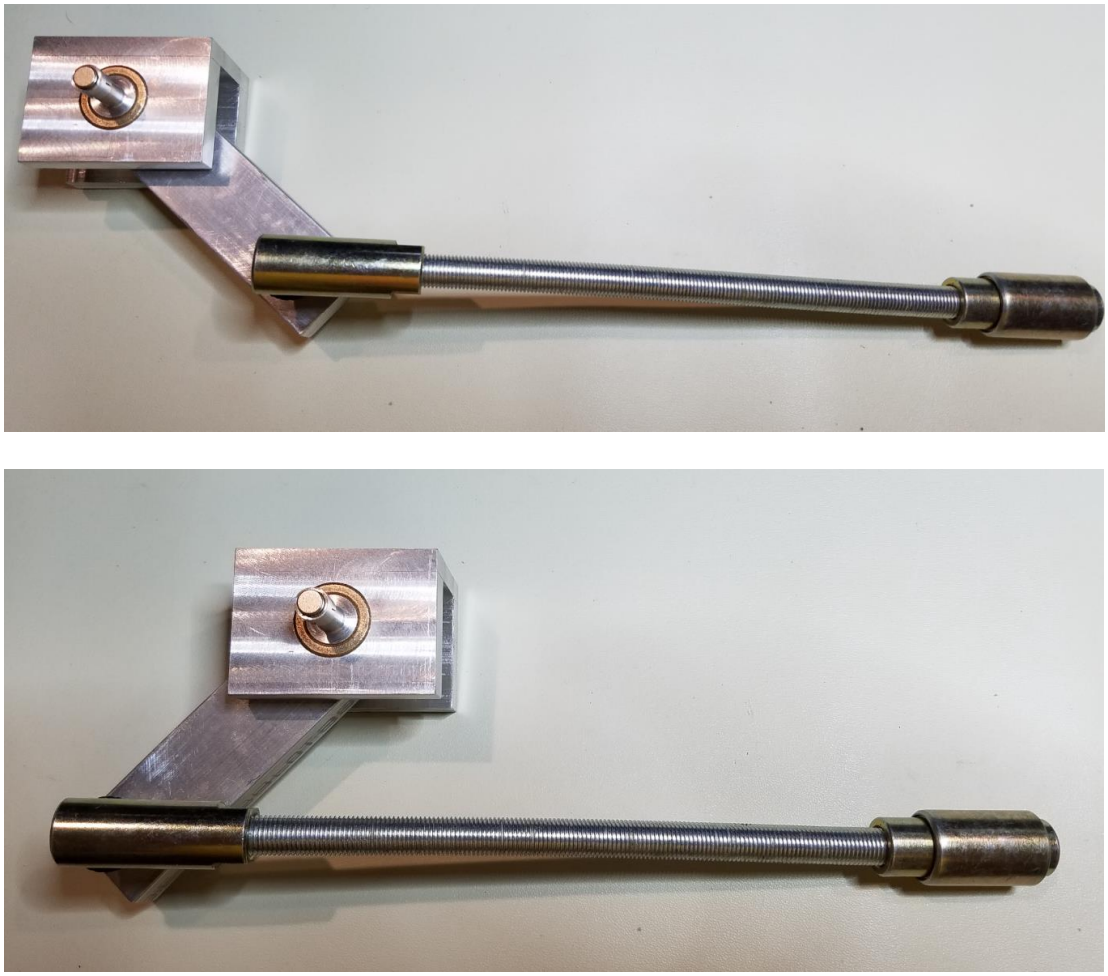


Figure 121. Testing the control arm sensor bracket range of motion

Once the control arm sensor bracket was made, I mounted it on the mower (Figure 122). Now when the control arm is moved, the control arm sensor bracket linkage moves in tandem. The absolute magnetic sensor picks up the rotation of the

magnet and outputs an analog voltage that is in direct relation to the angular position of the shaft.

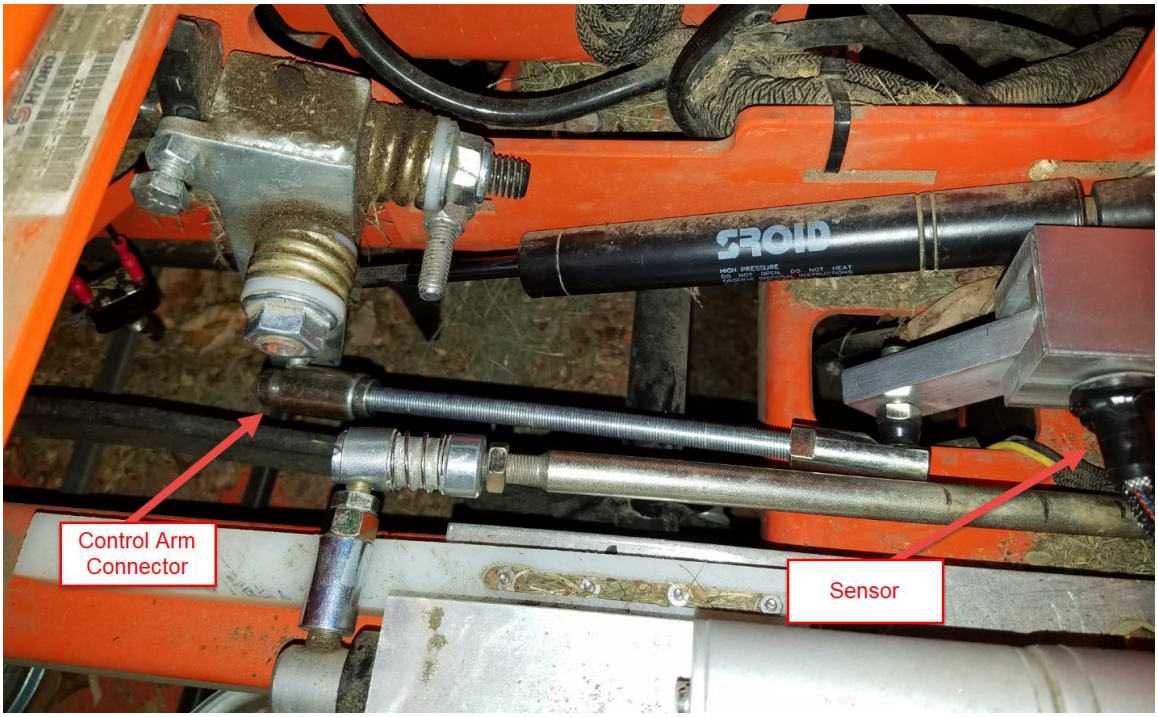


Figure 122. Control arm sensor bracket mounted

The output of the absolute encoder is connected to an analog input line on the Teensy 3.2 microcontroller used for the control panel. I did this for both the right and left control arms. The analog input lines were then connected to a 10-bit analog-to-digital converter thereby providing up to 1024 possible positions for each arm to be in.

6.2.2 Software

The software side of the control arms was straightforward. Part of the main loop of the Control Panel Teensy 3.2 microcontroller was dedicated to reading the state of the Absolute Encoders through the analog input pins and emitting a message for each control arm if the current value was different from the previous value. I did apply a +- tolerance to the current reading so the current reading had to change by more than the tolerance

value before pushing out the Control Arm Change ROS message containing the control arm index and new control position.

6.3 Debugging

Debugging complex systems such as that present in the ASMO project can be an arduous undertaking. Many of the concepts that promote code reuse and maintainability, such as componentization and a message-based architecture, also make it difficult to debug when the system behaves in unexpected ways. Fortunately, the ROS environment provides a suite of tools to assist with the debugging and development process. This section outlines the programs I found useful for this project.

6.3.1 rostopic

The rostopic program allows insight into the various Topics currently running in the active ROS environment. As new nodes are started, any new Topics exposed by the new nodes are available for viewing. I frequently used the list command for rostopic to get a list of the currently available topics. In order to actually view the messages, I could use the echo command with rostopic. Figure 123 shows messages displayed from the imu Topic during a live test of the system.

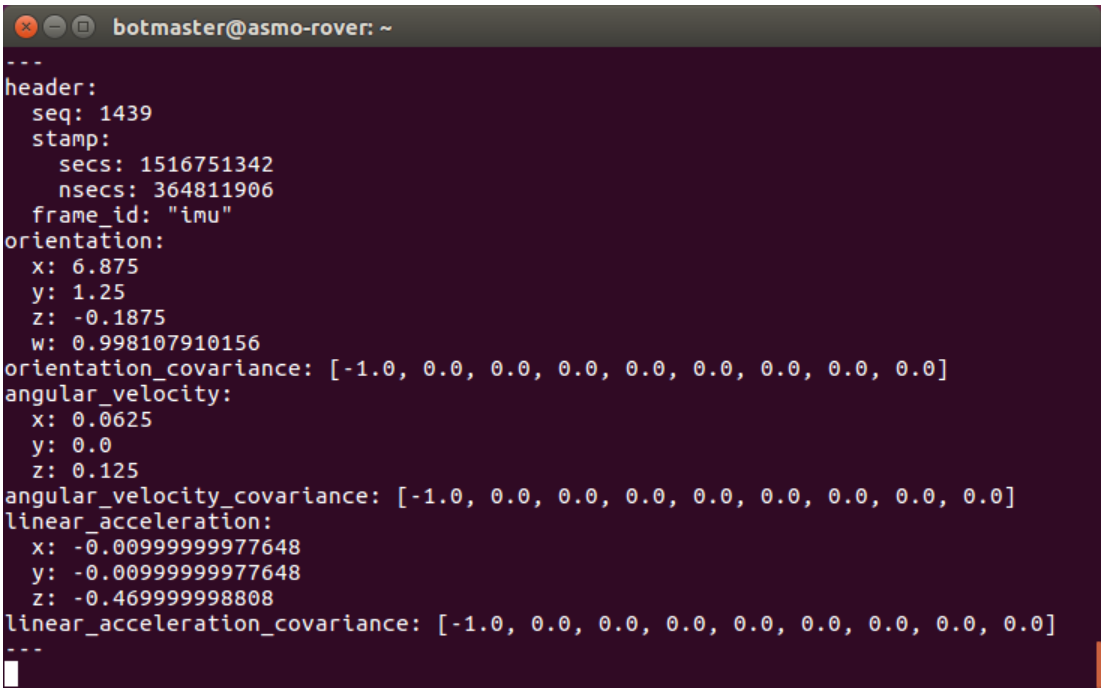


Figure 123. rostopic using the echo command to show the IMU messages

6.3.2 rqt_console

The rqt_console program is a GUI program that displays messages logged in the ROS environment. There are various log types that can be captured in ROS, such as Information, Warning, and Error log messages. These messages are written to the rosout Topic using the ROS_DEBUG function in a C++ ROS node or using the rosinfo function in ROS Serial nodes. The former is used in ROS nodes present on the main CPU while the latter are used in Arduino programs using the ROS Serial library. The use of ROS logging allows for a common, centralized way of handling debugging messages. Figure 124 shows the rqt_console program during the startup of the Radio Box. This process provides the equivalent of writing to stdout in a C/C++ program.

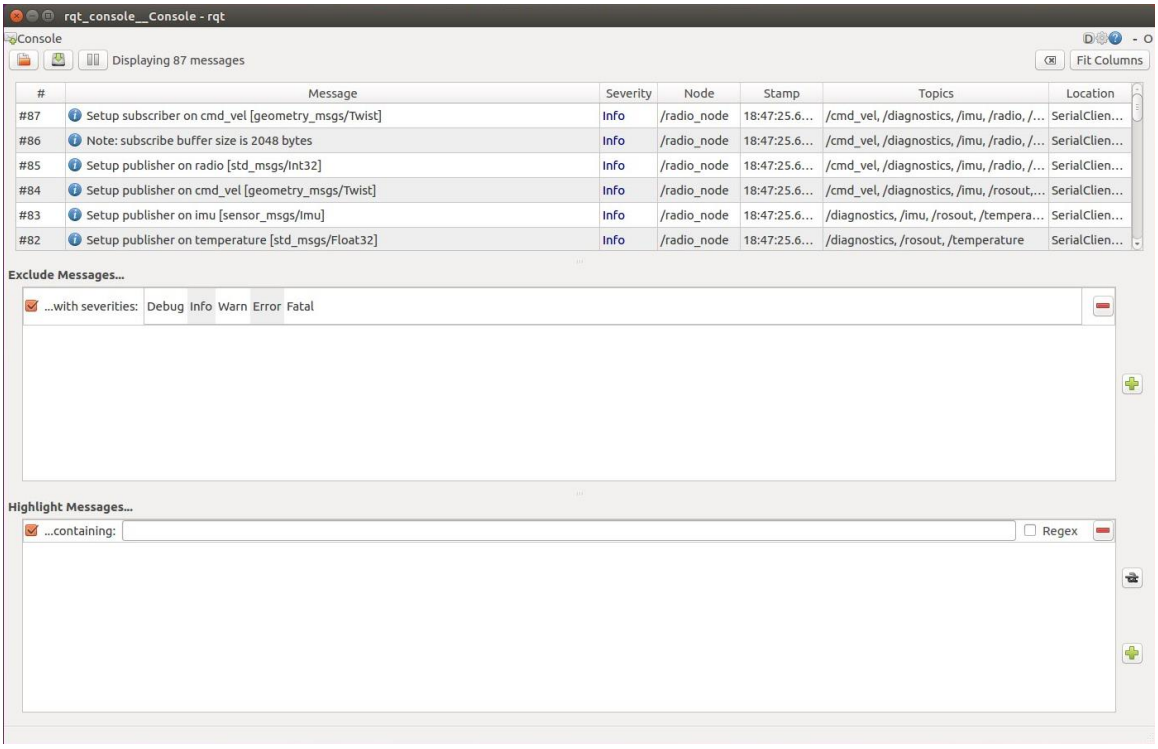


Figure 124. The rqt_console program during the Radio Box startup

6.3.2 RVIZ & rosbag

The RVIZ program is by far one of the most useful debugging and development tools available in ROS. It allows the sensor data in ROS to be visualized in relation to either a fixed point in the world or in relation to the robot itself. Figures 90 and 92 are perfect examples where this visualization comes in useful. It would be very difficult to make sense of the laser data simply by looking at the range data coming in from the LIDAR units. However, RVIZ is able to visualize the laser data by plotting it in relation to some fixed or relative point. This capability was also used extensively during odometry testing as Figures 116 and 117 demonstrate.

The ASMO system generates a great deal of sensor data. This sensor data gets emitted throughout the ROS environment as messages being published over Topics. The

rosbag program provides a mechanism to record messages being sent around in a running system. The messages can be saved to a file and then played back later. The RVIZ program does not care whether the information being rendered is live or is being played back from a data file created with the rosbag program. This is also the case with the other tools in the ROS environment. The ability to record sensor data while the robot is operating and play it back later was very helpful when working through various odometry algorithms. I was able to record the wheel encoder messages along with the IMU sensor messages and then play them back later while testing various approaches to sensor fusion. This ability to capture sensor data and work with it offline allowed me to continue refining my approaches even when the weather would not cooperate for field testing.

6.3.2 Remote Access

Another feature of ROS that helped with the development and debugging process was the ability to utilize all the aforementioned tools from a remote workstation. ASMO runs Ubuntu on its Main CPU Box. This Main CPU Box was connected to my home's Wi-Fi network. I was able to SSH into the Main CPU Box from a laptop for purposes of interacting with Linux and ROS from the command line. However, I still needed a way to remotely access the logs, visualize the sensor data, and view the various messages present when the mower was running tests. I was able to accomplish this because ROS allows for one install of ROS to work through another install of ROS by setting the ROS_MASTER_URI environment variable on the local install (on the laptop) to the IP address of the remote install (on the robot). Once this had been done, all the ROS tools worked with the remote install of ROS as if it were local. This approach allowed me to work from my laptop as if I were sitting on the mower with a keyboard and monitor on

my lap, which was the approach I had to take until I was able to get this network support figured out.

6.4 Training Mode

There are currently two steps that are a precursor to ASMO mowing an area autonomously. The first is defining the work area and the second is building a map of the work area. These steps are performed manually by an operator. Defining the work area is done with training mode turned on. In order to turn training mode on, the Autonomous rocker switch must be turned on. Once this is done, pressing the button directly below the Autonomous rocker switch turns on training mode. This button can be seen in Figure 118(4). When this button is pressed, it sends a TrainingModeStart message, to which the custom ROS node called Foreman is subscribing. When the Foreman service receives this message, it resets the IMU to zero out the current heading, captures the current GPS coordinates, and then begins to capture sensor message data.

The Foreman node turns on message capturing using the programmatic interface to rosbag. The programmatic interface to rosbag works in a similar way to the command line interface. The bag file created during this process is then used during the Map Making process outlined in the next section. Once training mode has been turned on, the operator drives the mower around the perimeter of the work area. Once the perimeter of the work area has been covered, the operator again presses the button directly below the Autonomous rocker switch. Doing so causes the Control Panel microcontroller to send out a TrainingModeStop message. The Foreman service receives this message and ends the capturing of the message data to the bag file.

6.5 Map Making

A ROS navigation map is a probabilistic representation of the world generated from sensor data. The sensor data used to generate the maps comes from the wheel odometry and the LIDAR units, which was stored in the bag file during the training process. ROS navigation maps are represented by a two-dimensional grid. Each cell in the grid contains a numeric value corresponding to the probability of the cell being occupied. Higher values represent a greater likelihood of the cell being occupied. By default, each grid cell in a navigation map represents 10 square centimeters. The cell is considered occupied if the value in it is more than 65% of the total range allowed by the image format of the map. The unoccupied threshold value is 19.6%. Values outside this range are considered "unknown" (Quigley, 1015). The threshold values are configurable, though I was able to use the default values successfully.

A ROS map of the work area can be built once the sensor data has been captured and stored in a bag file. In order to build the map, the ROS gmapping_slam node must be started, along with the mapserver node. Once these two nodes are running, the rosbag program can be used to playback the sensor data captured in the bag file during the training process. The map built by ROS from the sensor data represents how ROS sees the world. Prior to reviewing the map built by ROS, it is important to understand what the map represents. Figure 125 shows a picture of the ASMO work area that was mapped by ROS.

Figure 125. Picture of the ASMO work area

Key areas of the picture have been numbered so they can be correlated with the LIDAR scan of the same area (Figure 126).

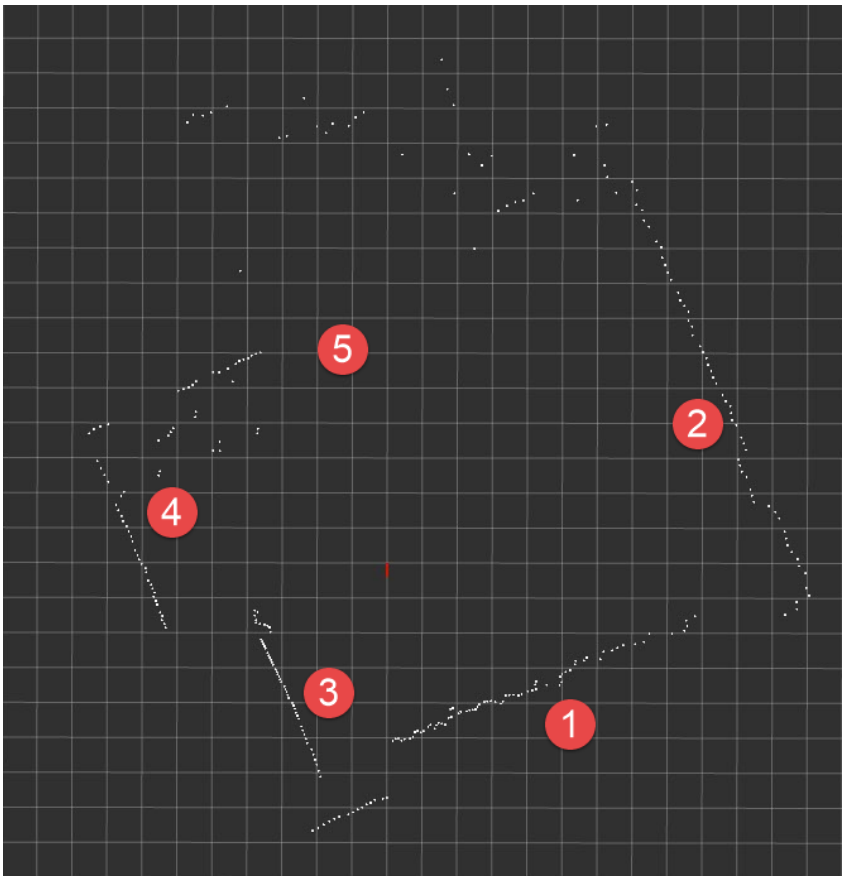


Figure 126. LIDAR scan of the work area in Figure 125.

Table 15 contains a list of the key areas and descriptions for the previous figures.

Table 16. ASMO work area feature description

Location ID	Description
1	A rock wall
2	A rock wall (occluded from the photograph as I was standing in front of it when the picture was taken).
3	Side of the garage
4	A large pile of rocks and gravel in the garage driveway in front of the garage opening
5	Bushes and other items that are randomly spaced alongside the road

ROS builds a map of the area using data from both LIDAR and Odometry messages. The ROS tf node transforms the location of the LIDAR unit with the center of the robot for purposes of correlation with the wheel odometry so messages from the tf node are also captured in the bag file. The transformation enables a common reference point for the coordinate system.

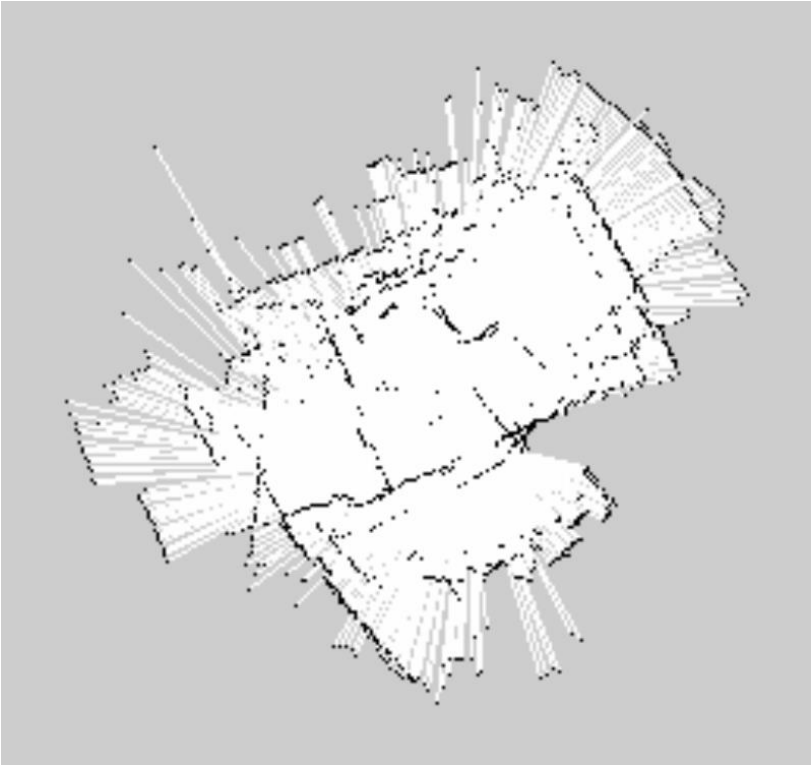


Figure 127. Initial map generated from minimal sensor data

Figure 127 shows a map that was generated from a single pass over the work area. The map in this figure is not very accurate. For instance, the side of the garage appears to be shown twice in the map (middle and left). In order to improve the map, I re-ran the training mode process and made multiple passes through the work area. Specifically, I drove the mower around the perimeter clockwise, then turned around and drove the mower back around the perimeter in a counter-clockwise direction. I also made a pass up

and back the center length of the work area. Then I created another map using the additional data captured. As shown in Figure 128, the second map generated with the additional data did result in a higher quality map. Performing these additional driving steps took several additional minutes during training mode, but I didn't find them to be overly burdensome. The training mode process would only need to be performed once for each new work area, not every time the yard required mowing.

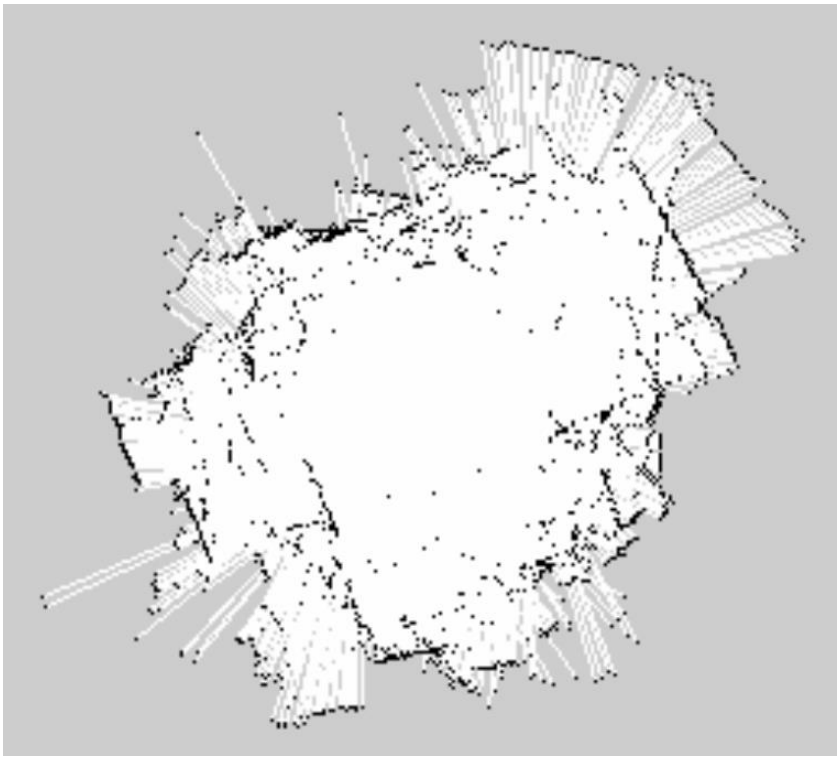


Figure 128. Map generated from more complete sensor data

Once the map has been generated, it is saved as a Portable Gray Map (PGM) formatted file by the map server node. The PGM format uses one byte per pixel to store greyscale information about the image. This map is then modified by the user in order to establish the work area boundary. This can be done using any image editing program capable of

reading in PGM files and saving in the Portable Pix Map (PPM) file format. The PPM format is the same as the PGM format except it uses 3 bytes per pixel to represent Red, Green, and Blue (RGB) color information. The work area should be established using straight red lines, as shown in Figure 129. The work area map can also be cleaned up at this point, removing any image artifacts that are known not to be obstacles. This file should be saved using the same name as the map file, but with a -workarea suffix. The nodes discussed in the next chapter use these two map files for localization and path planning.

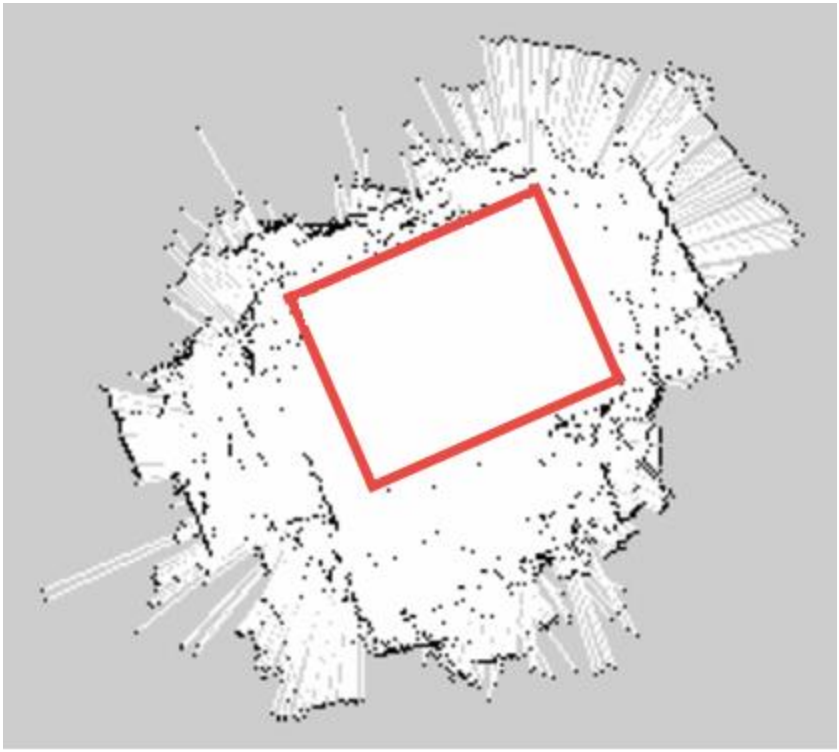


Figure 129. Established work area boundary

Chapter 7

Localization and Path Planning

The end goal of the ASMO project is to create a robotic system capable of autonomously mowing a yard. In order to carry out this task, ASMO must be capable of determining a path that effectively covers all of the work area in an efficient manner. This means not performing random mowing patterns as seen in most commercial autonomous mowers today, but instead following the more aesthetically pleasing parallel line patterns used by most landscapers. The ROS software used in this project does provide a variety of navigational services, but these are geared more towards the loose navigation from one point to another while avoiding obstacles, not towards the precise coverage navigation needed for mowing a yard. To achieve the goal of efficiently and precisely mowing a yard, I implemented a new path planning node in the ROS environment. This path planning node creates an in-memory virtual model of the work area which is then used to determine the best path to follow. Execution of this path is then carried out using navigational waypoints in the ROS environment.

An important precursor to utilizing any map effectively is knowing one's current location within the map. This is also the case for an autonomous robot. The process of determining the position of a robot within a map is known as localization or pose estimation (Choset, 2005). The problem of localization is that there is usually not a direct error-free way to sense a robot's position. Instead, sensor data from the odometry system is combined with sensor data from the LIDAR system in order to determine the position on the map that has the highest probability of being the location of the robot. This process occurs in real-time as the robot moves through space, so the system can make

compensating adjustments to the drive mechanism to keep the robot on the path previously determined by the planner. Knowing the location of the robot in real-time is also required in order for the robot to know when a goal has been reached.

7.1 Localization

The type of localization necessary for the ASMO project is known as position tracking. Position tracking is a local, as opposed to a global, localization problem. Global localization problems involve an initial starting position that is unknown. In the ASMO project, the starting position is established by the operator during the training process, so it is known. This was primarily done for safety reasons in order to place the robot into a known bounded area. Given the known starting location, the uncertainty involved in tracking the position of the robot over time is confined to the local region, near the robot's true pose.

Another assumption for the type of localization required in the ASMO project was that the environment stayed relatively static. Meaning, the map was known ahead of time and the only uncertainty involved in computing the robot's position was that of the robot itself. The intention was to keep any dynamic obstacles out of the work area during the mowing process. While this was the working assumption, the ability to detect and avoid dynamic obstacles such as people and animals was handled for safety reasons. However, while these obstacles did need to be detected, they did not need to be modeled from a localization problem perspective. This was due to the stop-and-wait approach taken to dynamic obstacles, which is discussed further in this chapter.

There are two methods to handling localization problems: active and passive. An active approach allows the localization algorithms to control the behavior and activity of

the robot. This approach means that the robot controller can actively seek new sensor data that would be relevant to disambiguating sensor readings. For instance, if a robot was located in an office hallway with multiple doorways that all appeared similar, the only way to disambiguate the various doorways might be for the robot to travel into one of the offices. The ability to actively explore an area is not suitable for a robotic mower where its job is to produce symmetric mowing patterns. Instead, the passive approach to localization is used in the ASMO project. This means instead of actively exploring an area, the localization is restricted to passive observation of its surroundings.

7.1.1 Adaptive Monte Carlo Localization (AMCL)

One of the most popular algorithms available for performing pose estimation is the adaptive Monte Carlo localization algorithm (AMCL). The AMCL algorithm allows for continued localization of a robot in a map using features and landmarks present on the map. The AMCL algorithm is implemented in ROS by the amcl node and is based on probabilistic state estimation, which uses a combination of laser data and odometry data to localize itself within a known area. In the case of ASMO, the area is mapped ahead of time so a work area boundary can be established by the user. Once established, sensor data from the IMU and wheel encoders are fused together to provide odometry data, which is then combined with laser data allowing ASMO to localize itself on the map. The ROS amcl node works by continuously matching laser scans to the map while detecting and compensating for drift in the wheel odometry data.

The ROS amcl node contains an implementation of a collection of probabilistic localization algorithms described in the book *Probabilistic Robotics* by Sebastian Thrun, Wolfram Burgard, and Dieter Fox (Thrun, 2010). Taken together, these algorithms

comprise what is known as adaptive Monte Carlo Localization. The amcl node works by maintaining a set of robot positions, called poses. These poses represent potential locations the robot may reside in. Each pose has an associated probability value assigned to it. In the case of ASMO, where the starting position is known, the initial pose has a high probability assigned to it of being the correct pose. As the robot then begins to move around, new candidate poses are created and assigned probability values. Higher probability values correlate with candidate poses that are more in line with map positions that can be matched given the current data coming from the LIDAR and Odometry systems while lower probability values correlate with candidate poses that do not match as well.

One issue I ran into using the AMCL approach to localization is that it required sufficient landmark points in the work area to operate effectively. The first set of testing I did with localization was in a large open field. This is where I had been testing the wheel odometry, so it was natural to continue additional subsystem testing in this area. However, in this open field there were not enough points of interest noticed in the LIDAR data for the localization algorithm to work. Even with a supplied starting position, the amcl node quickly lost its position lock as the number of probable pose locations quickly increased. For this reason, I moved the test area from the open field to an area closer to my house that had a larger array of structures, walls, bushes, and other interesting features for the localization to use as landmark points in the map.

It is important to note that AMCL represents a probabilistic approach to localization. Meaning, it does not reflect the exact position of the robot but rather the pose with the highest probability that is very likely to closely reflect the actual position of

the robot. As pointed out by Quigley in his book *Programming Robots with ROS*, "In practice, this means that when you use the navigation system to move the robot to a particular place in the world, it's likely to get close, but it will never end up in exactly the right place, even if the localization system claims that it's there" (Quigley, 2015). It is slightly unsettling to realize that a 27 HP, 1500 lb robot may not be located exactly where it thinks it should be located. However, this is the best situation given there are no sensors that can precisely determine the robot's position. GPS is used to locate the robot in the world, but it is only accurate to 9 feet. As previously mentioned, GPS was used as a safety system to prevent the robot from moving outside the work area. In an area populated with sufficient landmark points for the localization to operate, the AMCL approach was able to retain the pose of the robot within 12 inches. This precision was taken into account during path planning, which is discussed in the next section.

7.2 Long-Term Path Planning

Long-term path planning in the ASMO project involved creating a navigational plan to cover the entire work area in an efficient manner. ROS has a built-in path planner, but it is geared towards navigating indoor environments such as office areas. The built-in planner it is adept at navigating from point A to point B but is not well suited for work area coverage, which is required for an autonomous lawnmower. Therefore, I developed a ROS planning node capable of creating paths to more fully traverse an area. Additionally, the custom planner does not create random paths, but instead designs paths that follow the typical back-and-forth pattern used by most landscapers. The planning node takes the work area map and converts it to a virtual in-memory model of the work area. The planning node also creates a virtual representation of the mower, which is

capable of moving around the virtual model of the work area. It then uses a Boustrophedon path planning approach to lay out navigational waypoints that cover the work area. Since the Boustrophedon decomposition of the work area could result in multiple sub-cells being created that must be visited separately, as well as planned for separately, the built-in ROS navigation system was utilized to travel between adjacent work area cells.

7.2.1 Coverage Grid

The work area map discussed in section 6.5 is a ROS navigational map that has been annotated with a red rectangle to establish the work area boundary (Figure 129). This map is stored as an image file with each pixel representing a 5 cm square area. For path planning purposes, the 5 cm resolution was too fine-grained and would result in a large grid that is difficult to manipulate and reason about. It also didn't establish a buffer to account for the size of the mower when describing obstacles and boundaries. Instead of trying to work with the original map file, I decided to create a new grid that could easily be represented as an in-memory model of the work area. Therefore, I chose to create a 2-dimensional in-memory array to represent the work area. This array is created on-the-fly from the work area image file when the planning node starts.

The size of each cell in the array is based on a quarter of the working width of the mower. The mower has a 54" cutting width. I used 56" as the working width since it is evenly divisible by 4 and very close to the actual cutting width. Each cell of the in-memory array therefore represented 14 square inches. I chose to divide the mower width by 4 as this seemed to represent the best compromise between a grid that was too fine grained and one that was not granular enough. The idea was that I did not want the

planning routine to create a path for the mower that came within one quarter of its working width, or 14", of any obstacle or boundary area. The localization routine was able to localize the mower within 12", so 14" offered a 2" margin on top of the 12" margin of error.

The coverage grid is created automatically at startup from the work area map, whose file name is supplied as a startup parameter to the node. The coverage grid node reads the work area map file into memory and scans the resulting array for any red pixel cells. Recall that the red pixel cells were added manually by the user when creating this file from the grey-map navigational file. It was stipulated during the creation of the work area map that the work area would be rectangular. This meant that once the corners of the rectangle were located, the size of the rectangle could easily be defined by subtracting the X, Y coordinates of two opposing corners. The desired corner to use as the origin point is supplied as a startup parameter. The corners of the rectangle are designated 1 through 4 starting with the upper-left corner and proceeding clockwise. The starting position and direction of the mower were also supplied as starting parameters of the coverage grid node. For purposes of my test area, corner 4 was chosen as the origin point.

The work area map used for testing purposes was a 40' (480") wide by 60' (720") long section of the yard. This meant the in-memory array needed to be 34 x 51 elements. The first dimension of the array represented the X axis of the work area while the second dimension represented the Y axis of the work area. Each element in the area could be one of four possible values (Table 16—Element Value). An advantage to representing the coverage grid as a small in-memory array is that it made debugging and visualization

very straightforward. I created a printMap function that I could use to render the current state of the virtual in-memory grid at any time using ASCII values (Table 16—Element Visual).

Table 17. Coverage Grid possible cell states and their visual representation

Element Value	Element Visual	Element Description
0	.	UNOCCUPIED
1	@	COVERED
2	X	OCCUPIED
3	#	MOWER
3	O	MOWER (wheel locations, determined via offsets based on the mower direction)

This state could then be dumped to the screen or a file using simple ROS service functions. Figure 130 shows the printMap result of the coverage grid created from the test work area file. Note that position (1, 1) represents (0, 0) of the 2-dimensional array. Also note the location and direction of the mower. The mower direction is visually represented by the location of the wheels. The cells containing the mower are represented by the constant MOWER, the location of the wheels is printed using the O character based on the direction of the mower. The wheels are located on both sides of the mower towards the rear. So, Figure 130 represents the mower facing the Northern direction. It should also be noted that North on the printed map does not necessarily represent the Earth's magnetic North. It represents North relative to the map itself, with

North being the first row of the 2-dimensional in-memory array while South is the last (51st) row of the array. So, element location (0, 0) is the North-West corner of the map.

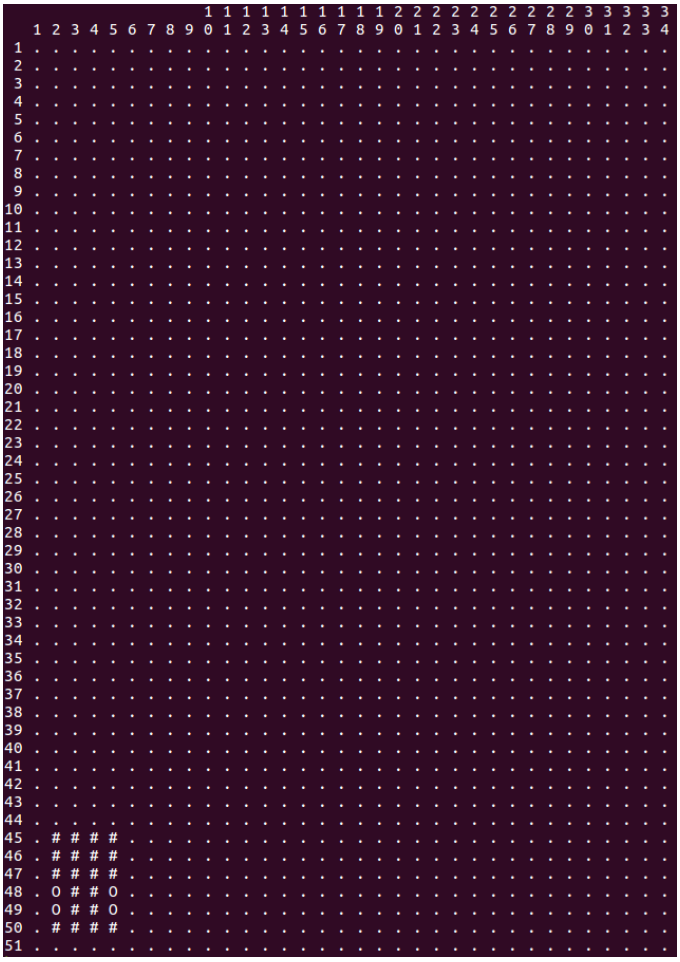


Figure 130. Result of the printMap function showing the initial state of the coverage grid

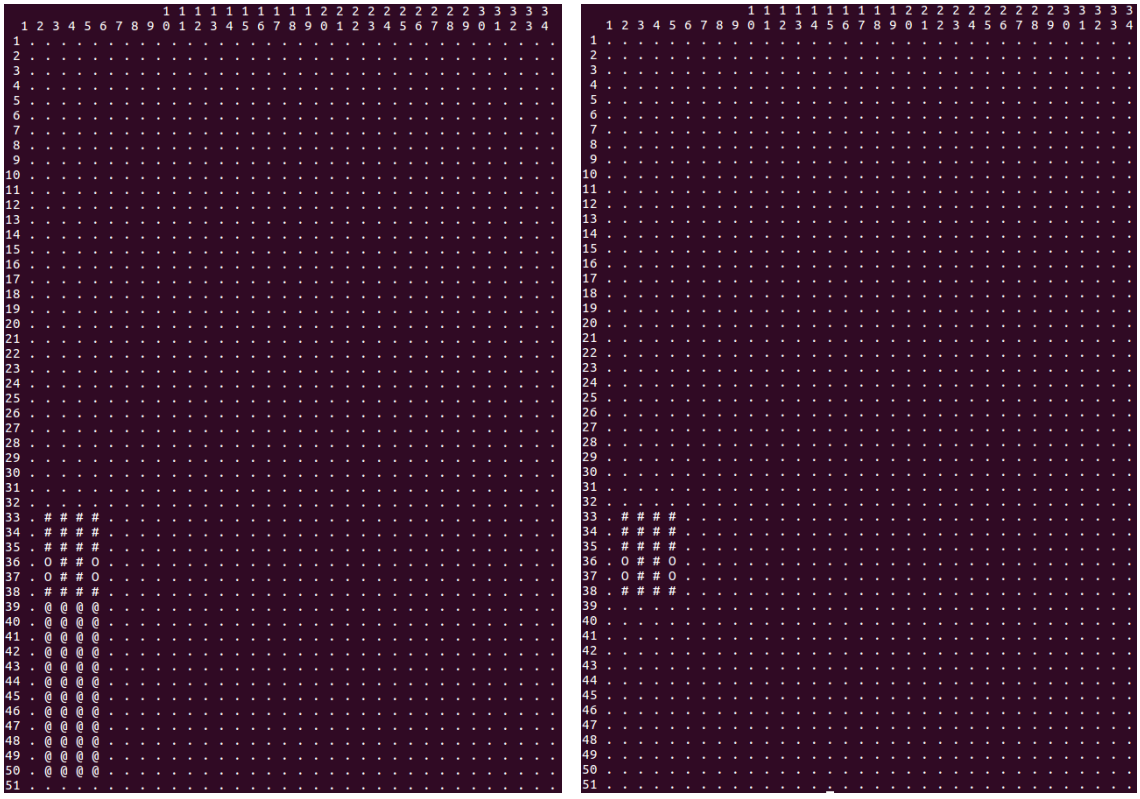
The state of the mower is represented virtually in this in-memory model. This means the virtual mower can be used to not only track the actual state of the mower but can also be used to explore possible paths the actual mower could later be made to take. In this way, the virtual model can be used to discover and plan paths. In order to manipulate the virtual mower, simple functions were created (Table 17). These functions could then be used to move the virtual mower around in the virtual in-memory work area.

Table 18. Virtual mower functions

Function Name	Function Description
moveForward	If able, moves the virtual mower forward one cell position and returns true. If unable, the function returns false.
moveBackward	If able, moves the virtual mower backward cell one position. If unable, the function returns false.
moveTo	Jumps the mower to the specified location. The mower blades are turned off if they are on. Returns true if the virtual mower was moved to the location, false otherwise.
turnRight	If able, adjusts the position of the virtual mower 90 degrees clockwise. If unable, the function returns false.
turnLeft	If able, adjusts the position of the virtual mower 90 degrees counter-clockwise. If unable, the function returns false.
bladesOn	Updates an internal flag indicating the mower blades are currently ON. With the mower blades on, when the virtual mower moves, its previous position in-memory is written with the COVERED value.
bladesOff	Updates an internal flag indicating the mower blades are currently OFF. With the mower blades off, when the virtual mower moves, its previous position in-memory is left as it was before the mower entered the position.

It is worth noting the reason for including the ability to turn the virtual mower blades on and off. When the virtual mower blades are on and the virtual mower moves, its current position before moving is updated with the value COVERED. The virtual mower's

position is then updated within the in-memory model of the work-area. In this way, the mower can be moved around the in-memory work area updating it to indicate whether the move included grass that was cut or not. Figure 131(a) and 131(b) show these two states after the virtual mower has been moved forward two positions from the starting point shown in Figure 130. One example of not cutting the grass during a move would be when traversing from one work area cell to another. In this case the mower cutting blades are turned off during this transition.



(a) (b)

Figure 131. (a) forward movement with the cutting blades on and, (b) forward movement with the cutting blades off

7.2.2 Planning Algorithm Description

The approach taken in the ASMO project for path planning follows the Boustrophedon cellular decomposition algorithm described by Choset (Choset, 1998).

The first step of this algorithm is to sweep a line through space stopping at vertices, which are called events. A list of these vertices, or edges, are tracked during the sweep. The space between the edges are known as cells. Each cell is then visited and covered using a back-and-forth motion—it is from this back-and-forth motion that Boustrophedon takes its name, as it is Greek for "way of the ox." Since my initial work area did not contain any obstacles to account for, this first test area is considered a simple cell. Therefore, I decided to implement support for the Boustrophedon path planning from the inside out. Meaning, I first started development of the algorithm by creating a program to cover a simple area using the back-and-forth motion pattern. I would then later expand the path planning capability to handle more complex scenarios.

The virtual mower can be moved forward by calling the moveForward function. If the function is able to move the virtual mower forward, it will return true, otherwise it will return false. The function cannot move the virtual mower forward if doing so will cause the new mower position to be outside the bounds of the 2-dimensional array. So, if the mower is facing North and moving forward, it cannot be moved past the 0 Y position. Given that I also wanted to include a buffer, the virtual mower is not allowed to move past the 1 Y position. Additionally, if the mower blades are on, the motion of the virtual mower is restricted from moving into an already mowed area. An already mowed area is defined by an area of the in-memory array that contains COVERED values.

Given these movement restrictions, determining the first leg in the coverage path became a simple matter of calling moveForward until it no longer returned true. Figure 132 shows the result of printMap after this had taken place. The mower had moved to the Northern most position of the coverage map. The moveForward command along with the

current X, Y grid location and the state of the mower blades are captured in a C++ class called PathCommand. The PathCommand instance is then stored in a vector called PathCommands. The PathCommand class was created to capture the movement and direction of the virtual mower so it can be later converted to movement commands for the physical mower.

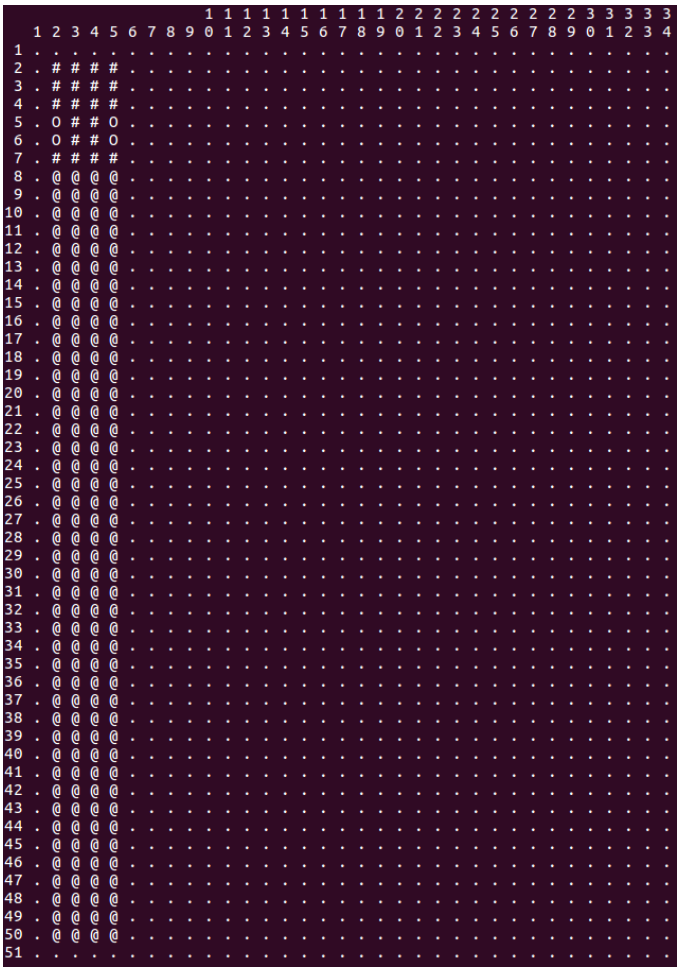
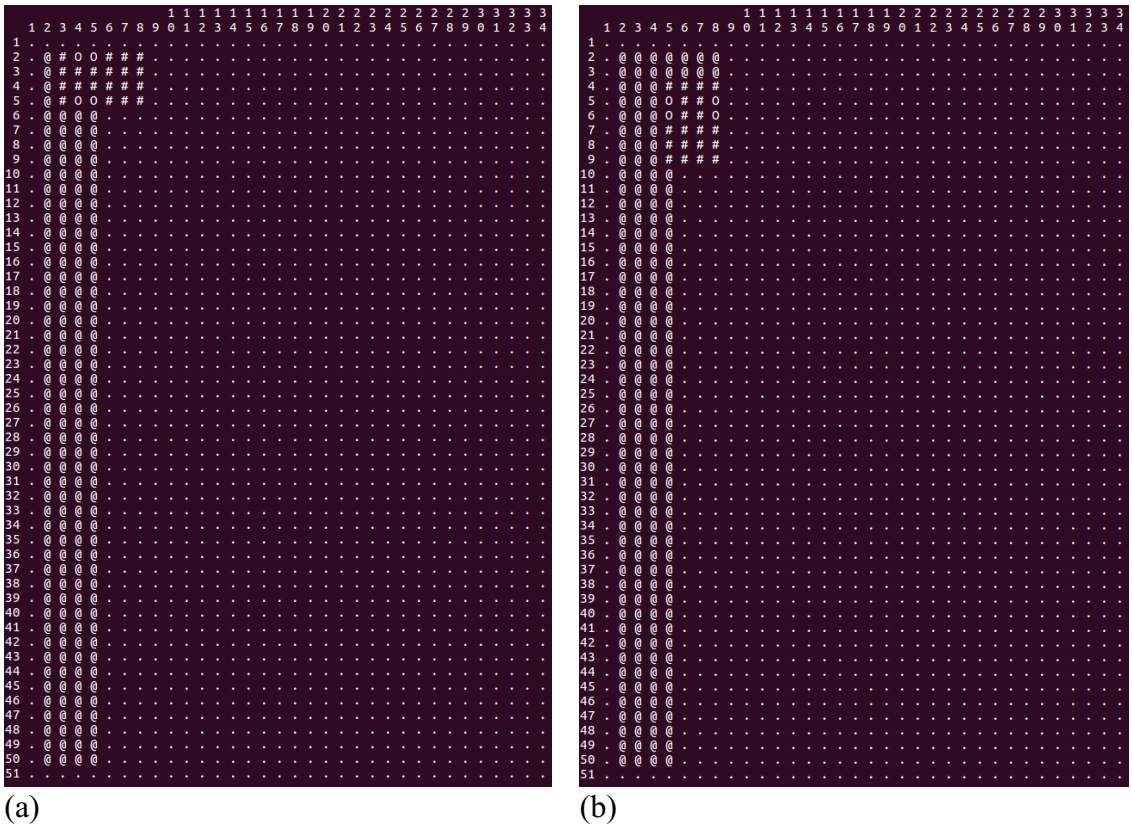


Figure 132. Virtual work area after first leg of path planning

At this point the planning algorithm needs to decide which way to turn the virtual mower—Left or Right. It turns in the direction that is farthest away from its current X position. In the current case, the virtual mower's starting position equated to (2, 45) in the in-memory coverage grid. Since the grid is 34 elements wide and the virtual mower's

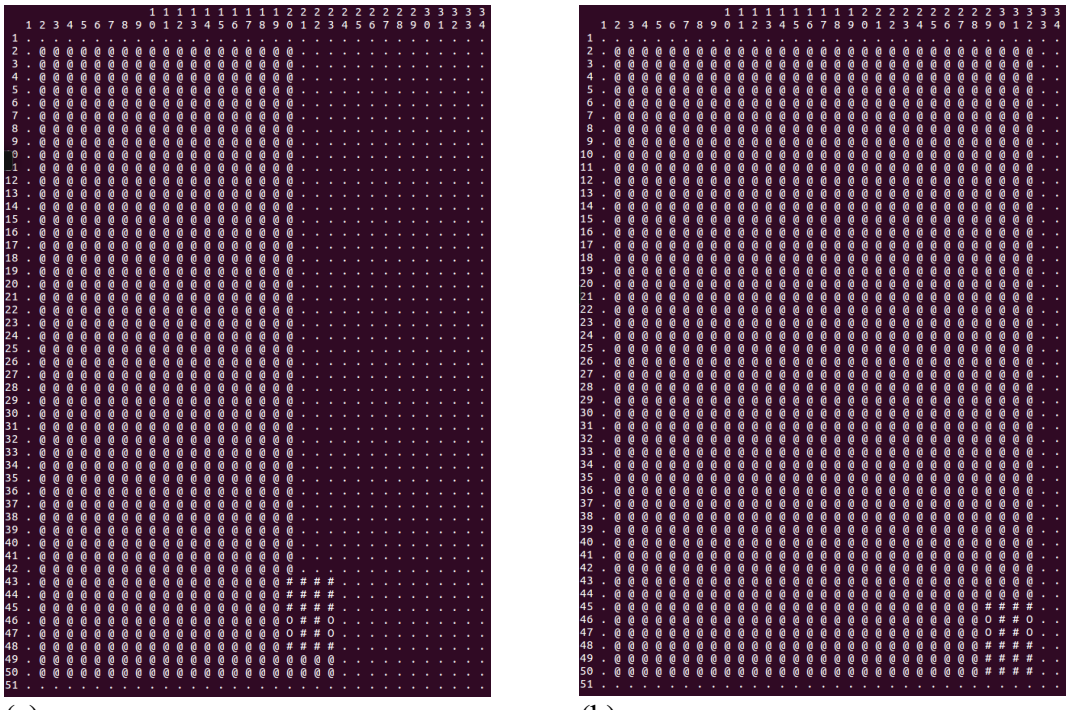
current X position is 2, the algorithm calls the turnRight function in order to turn the virtual mower to the right. Since the turnRight function only turns the mower 90 degrees, the algorithm calls the turnRight function twice in order to turn the mower around to face the direction it came in order to follow the Boustrophedon path. It should be noted that the turning operations cause the mower to pivot around the wheel on the side of the direction of the turn. This pivot operation results in the mower facing the direction it just came from, but positioned over one mower-width, minus one wheel-width, as shown in the Figure 133(b). The result of printMap after making these two calls is shown in Figure 133. Two PathCommand instances are stored in the PathCommands vector capturing these moves. Additionally, the turn functions capture the current turn direction and store it in a variable called lastTurnDir for use when making the next turn.



(a) (b)

Figure 133. The coverage grid after (a) the first call to turnRight and (b) after the second call to turnRight

Turning operations include an overlap of the mower position. Since the virtual mower is 56” wide and the actual mower is 54” wide, some overlap was required. A one-cell, 14”, overlap also accounts for slight deviations in the mower movement as it moves along the path. Next, the planner proceeds to make additional calls to moveForward until it can no longer do so. With the mower now facing South, calls to moveForward result in the virtual mower moving down the virtual work area until it reaches the Southern boundary. Once at the bottom of the work area, calls to turnLeft are made in order to re-orient the virtual mower towards North. The turnLeft function is called because the lastTurnDir was RIGHT and the routine alternates turn direction in order to cover the work area. This process continues until turn functions are no longer able to turn the mower. At this point, coverage is complete for this basic test work area. A breakdown of this process is shown in Figure 134.



(a) (b)

Figure 134. The coverage grid after multiple calls to moveForward and turnRight. The first image (a) shows the result of a call to printMap in the middle of the process while the second image (b) is the result at the end of the process.

Once no further movement is possible given the movement constraints, coverage for the cell area is considered complete. The vector of PathCommand instances now contained a list of all the edges encountered during the coverage process as well as the commands necessary for navigation. Note that there is a 14" buffer left around the boundary of the work area. It should also be noted that the current coverage process does leave an additional single 14" row along the right side of the work area. The main goal at this point was to establish a general routine capable of coverage that would transfer well to the physical mower. Once the virtual model was working with the physical environment, I planned to come back and implement additional passes to clean up the smaller remaining sections. This type of trim work is not uncommon in landscaping and would be required for closer cuts around objects such as trees and buildings.

7.2.3 Multiple Work Area Cells

The test work area described in the previous section represents testing performed on the actual mower. I had planned to do additional testing on the actual mower with more complex areas of the yard, but winter weather set in before I was able to do so. That said, I did implement the capability in the process to handle more complex scenarios. Complex scenarios are defined in this context as more than one cell that requires traversal in order for coverage to be complete. There are various ways that additional work area cells could be created. The Boustrophedon algorithm describes cells as areas divided by an edge. In the simple example previously described, the edge is the work area boundary itself. Additional edges could be introduced by adding obstacles to the work area. For this purpose, I added a new function to the coverage grid node called createObstruction. This function takes starting X, starting Y, width and height

parameters. It then marks all elements within this area of the in-memory grid as OCCUPIED.

The previous virtual work area was modified by calling createObstruction in order to add an arbitrary obstruction for testing purposes. Figure 135 shows the result of printMap after adding this obstruction.

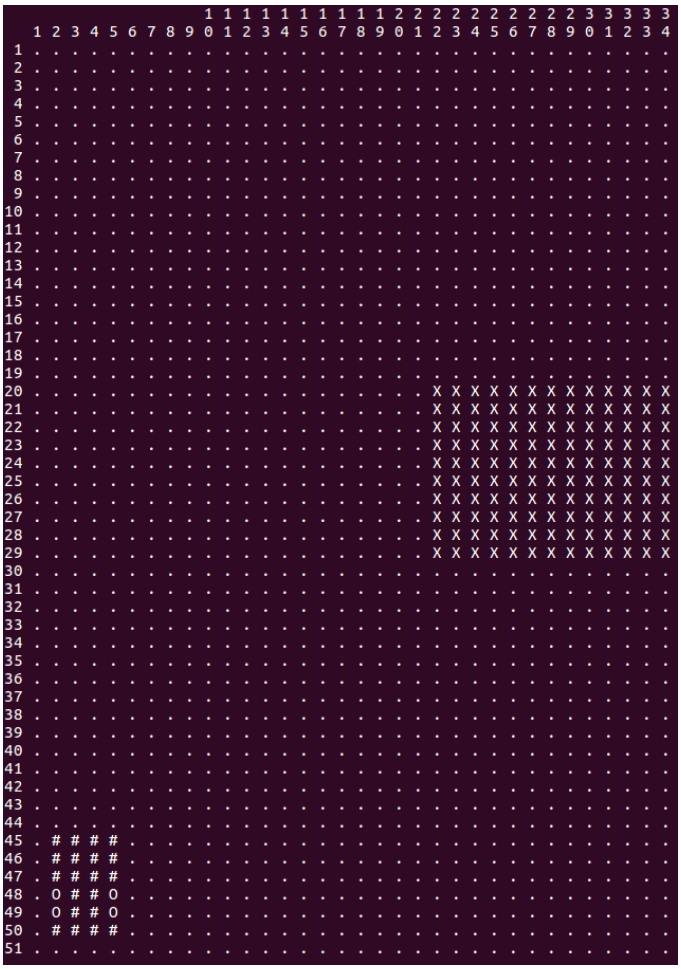


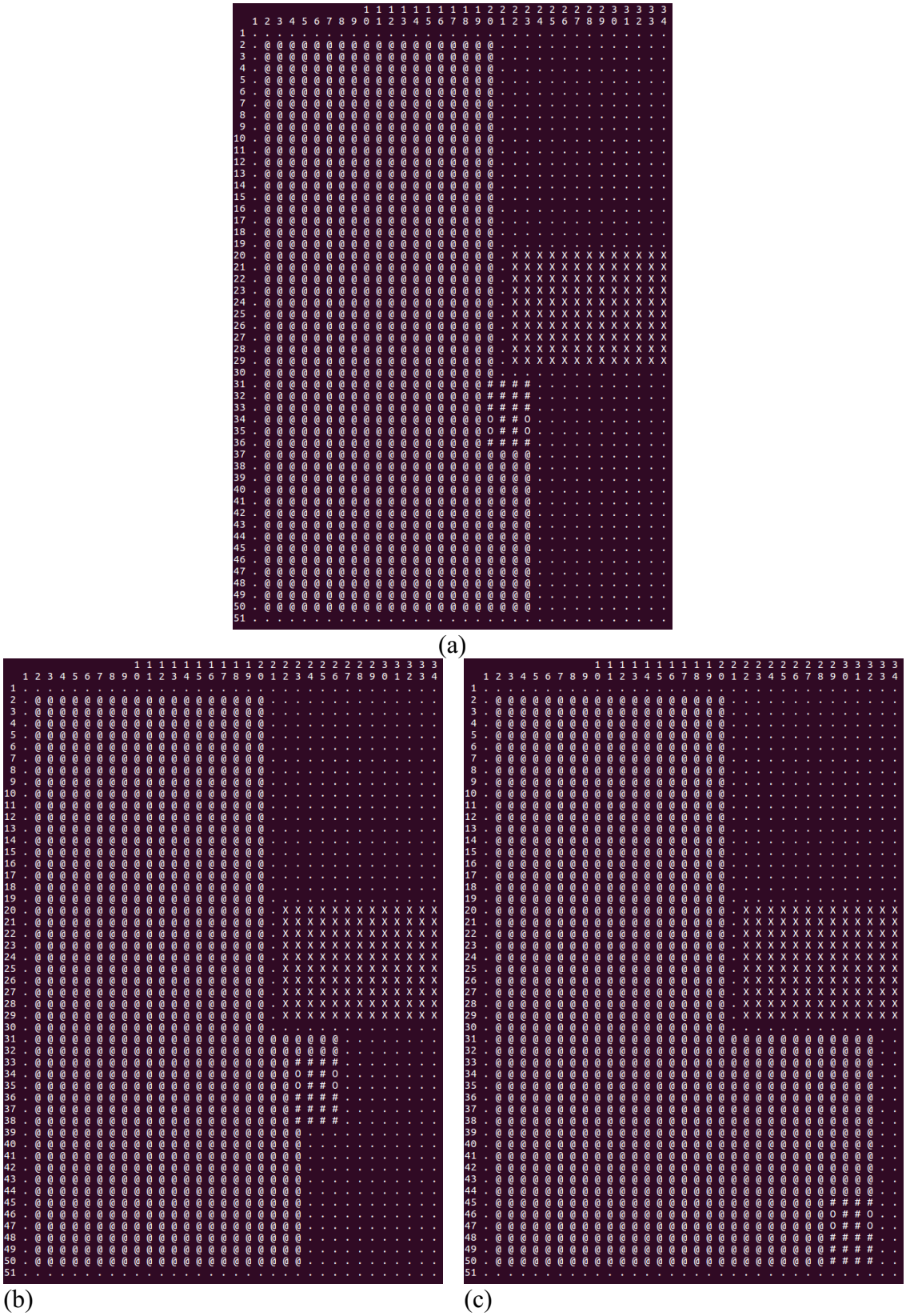
Figure 135. Virtual work area with an arbitrary obstruction added

In reality, most obstacles would not present themselves as nice rectangles. However, the idea behind the use of rectangles is that a process could be run on the work area map image file to automatically create bounding boxes around obstacles present inside the work area boundary. One such process is the Open Source Computer Vision Library,

otherwise known as the OpenCV project (OpenCV, 2017), which provides a set of libraries for performing this type of operation. The path planner would create paths around these obstacles and then once the major mowing operation has been completed, another routine could come back through to perform the trim work, tidying up areas around obstacles that required it. This tends to be how I personally operate the mower when mowing manually.

Once this obstruction was added to the virtual test area, the path planner was re-run. Figure 136 shows the behavior of the path planner as it encountered the new obstacle. Figure 136(a) is the point at which the virtual mower first encountered the obstacle. This location was recorded in an instance of the Point class, which captured the X, Y coordinates of the virtual mower position, as well as its direction, when it first encountered the obstacle. This Point class instance was then added to a vector called Obstacles.

Figures 136(b) and 136(c) show the virtual mower as it continued the path planning to the completion of the current work area cell. The original Boustrophedon algorithm would have created three cells out of this new test work area. My approach automatically merges two of the areas as part of the planning process based on the current direction of movement of the virtual mower. This approach leaves just one area that requires coverage.

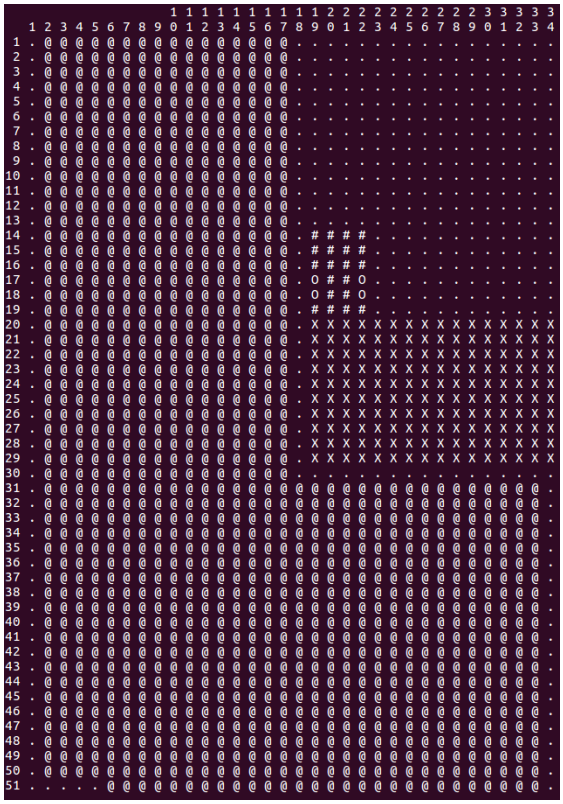


(a) (b) (c)

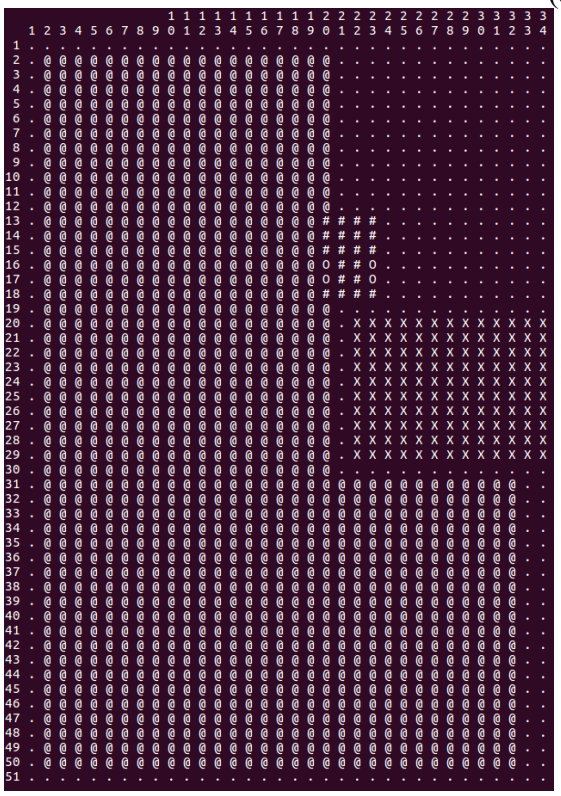
Figure 136. Path planner behavior as it encountered the new obstacle

Once coverage of the current work area cell completed, as depicted in Figure 136(c), the path planner iterated over the Obstacles vector. It did this by calling the findNextCell function. This function returns the in-memory grid coordinates of the next traversable work area. It does so by retrieving an item from the Obstacles vector and then moving forward from that location until either an area large enough to accommodate the virtual mower is encountered or the edge of the overall work area is encountered. If a suitably sized UNOCCUPIED area is found, the coordinates of this position are returned as an instance of the Point class. The direction of the mower is then updated based on the direction passed back in Point and then moveTo is called with the Point.X and Point.Y values. The call to moveTo records the movement as a PathCommand in the PathCommands vector.

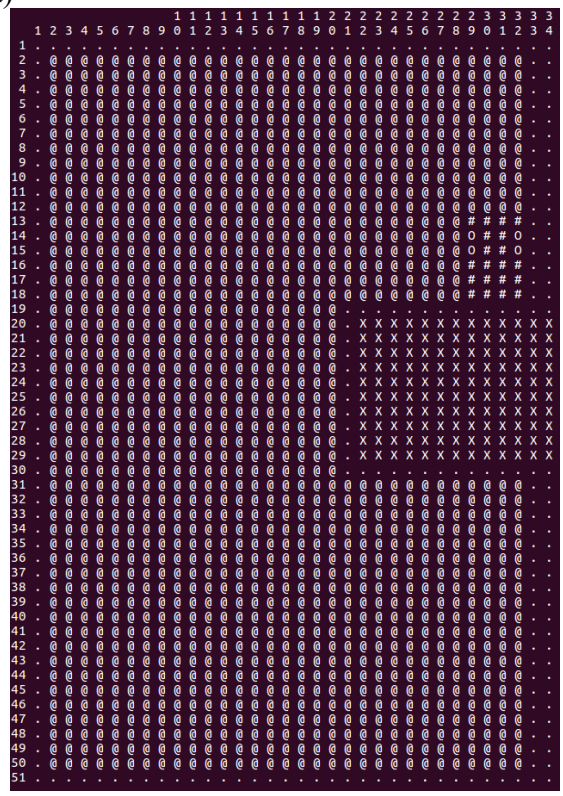
The use of the moveTo function causes the virtual mower to jump to the location on the virtual in-memory grid. When this function is applied to the physical mower, it would be done so using the ROS navigational capability. In this case, the mower needs to be moved from one point to another point while avoiding obstacles. As previously mentioned, this is the very type of activity ROS excels at, so it is made use of when needing to travel between work cells. Once the virtual mower is moved to the new virtual work cell, the path planner once again starts calling the moveForward and turn functions. As with the first work area cell, when any grid elements are encountered other than UNOCCUPIED, the planner reverses direction. Throughout this process, PathCommand points are added to the PathCommands vector, recording the steps necessary to execute the path. This process continues until no further movements are possible. This process is detailed in Figure 137.



(a)



(b)



(c)

Figure 137. Path planning behavior during work area cell 2 coverage

7.3 Short-Term Path Planning

Once the long-term path had been planned, it needed to be executed. The long-term path was executed as a series of short-term goals in the ROS system. This was done because the ROS navigation system can be difficult to predict if not tightly controlled. Its job is to find the optimal path from one point to another and the path it will take is not always the most apparent one as there are many variables involved in its decision-making process. This can be beneficial in environments where dynamic obstacles present themselves with regularity and therefore the navigation system needs to respond accordingly. However, in a mowing environment, it is the journey that is important, not simply the destination. As such, any unplanned movement outside a precise path tends to be counter-productive. It would not create aesthetically pleasing cuts to have the mower wander around the yard trying to avoid an obstacle that presented itself. This would also not be the safest approach to take, given the size and power of the mower.

The coverage grid planning node is responsible not just for creating the long-term path but also for executing it. To enable execution of the planned path, a service function was created called executePath. This function executes the last path that was created in memory. This function can be called manually from a terminal window or it can be called using the red execute button on the remote-control unit. To remotely execute the planned path, for safety reasons, the Autonomous Mode switch must be enabled on the operator control panel, as shown in Figure 138(1). When this switch is turned on, pressing the red execute button on the remote-control unit, as shown in Figure 138(2), results in the executePath function being called on the coverage grid planning node.

Figure 138. To execute a navigation path remotely, (1) enable autonomous mode and then, (2) press the red execute button.

7.3.1 ROS waypoint navigation

As previously mentioned, the ROS navigation system is very good at navigating from one point to another. It will also take the most direct route between these points if given the opportunity. The coverage grid planning node makes use of the ROS navigation system by producing short-term waypoints from the long-term planned path. It exposes its path execution capability publicly through a custom ROS service function called executePath. The executePath function iterates through the PathCommands vector turning each PathCommand into a ROS navigation goal using the custom createGoalFromPath function. This function handles transforming the in-memory grid coordinate system back into the navigational grid coordinate system used by ROS. This function also can issue messages to turn on or off the mower deck based on whether the move command requires cutting action. Recall that some waypoint navigation, such as moving between work cells, do not require the cutting deck to be active. Executing a long-term path means iterating through the PathCommands vector created during the path planning process and issuing waypoints to the ROS navigation system.

Each waypoint is issued to ROS individually. The waitForResult ROS function then blocks for the supplied duration before returning. A one second timeout is passed to

this function, which allows the function to timeout if the waypoint has not yet been reached. If this is the case, the function returns false, providing an opportunity for the coverage grid planning node to perform additional checks to determine if navigation should continue. If the coverage grid planner decides navigation should not continue, it cancels the current goal and exists. Navigation could be cancelled for reasons such as an unmapped obstacle being detected in the robot's path. Navigation is cancelled at this point to prevent the ROS navigation system from attempting to avoid the obstacle, which would result in the mower moving outside the planned path. The basic navigation logic is shown in Figure 139.

```
int errorCondition = NO_ERROR;
bool continueLongTermNavigation = true;
int index;
if (_stopWaitIndex >= 0)
    index = _stopWaitIndex;
else
    index = 0;

vector<PathCommand> pathCommands = planner.GetPathCommands();
while (continueLongTermNavigation)
{
    PathCommand pc = pathCommands[index++];
    move_base_msgs::MoveBaseGoal waypoint = createGoalFromPathCommand(pc);
    moveBaseClient.sendGoal(waypoint);

    bool continueShortTermNavigation = true;
    while (continueShortTermNavigation)
    {
        if (!moveBaseClient.waitForResult(ros::Duration(ONE_SECOND)))
        {
            if (obstaclesPresent(waypoint))
            {
                // an obstacle was detected, cancel both long and short term navigation
                continueShortTermNavigation = false;
                continueLongTermNavigation = false;
                errorCondition = DYNAMIC_OBSTACLE;
                ROS_INFO("Dynamic obstacle encountered...");
            }
        }
        else
        {
            continueShortTermNavigation = false;
            if (moveBaseClient.getState() != actionlib::SimpleClientGoalState::SUCCEEDED)
            {
                // couldn't meet the short term goal, cancel the long-term goal
                continueLongTermNavigation = false;
                errorCondition = GOAL_NOT_MET;
                ROS_INFO("Goal not met for some reason...");
            }
        }
    }

    if (index >= pathCommands.size())
        continueLongTermNavigation = false;
}
```

Figure 139. The navigation control logic for the executePath function

If the long-term navigation loop exits without an error condition being set, then the long-term goal has been met. Otherwise, either an obstacle was detected, or a short-term goal could not be met. If an obstacle was detected, the navigation goal is canceled using the MoveBaseClient function called cancelAllGoals and the stop-and-wait logic is initiated. The stop-and-wait logic is discussed below in section 7.3.3. If a short-term goal could not bet met, then execution stops completely.

7.3.2 Dynamic obstacle detection

The obstaclesPresent function used in the executePath function checks the LIDAR units for any *unexpected* obstacles within the immediate path of the mower. An unexpected obstacle is an object in the mower's path that is not accounted for in the navigational map. The long-term path planner has created a path to avoid any known obstacles present on the map. This means if an object appears in the path of the mower ahead of the next waypoint, it is a dynamic obstacle. There will be objects detected by the LIDAR units that will be in front of the mower, but these are only relevant if they appear in front of the mower and between the location of the mower and the next waypoint. This is the reason the waypoint is passed into the obstaclesPresent function in Figure 139.

This particular issue also serves to emphasize the importance of the localization routines. If the robot is not well localized, then it cannot determine whether or not an obstacle is actually present in its path. This is only an issue when the robot is approaching a waypoint. The waypoint was created in order to avoid a boundary point. The boundary points are usually not arbitrary but correlate with some particular feature in the physical world—such as the side of a building or stone wall. As the robot approaches

the boundary, the robot is going to pass within one grid cell, or 14", of the boundary—this is the buffer margin established in this project for objects. This 14" boundary may actually be as little as 2" due to the margin of error present in the localization routines—recall they have been tested accurate to within 12". During my testing, I introduced several obstacles of varying sizes from 12" in width to 36" in width and these objects were detected successfully. This was the case even within a few inches of a boundary edge. Figure 140 shows a LIDAR scan of an obstacle detection test where a 24" square cardboard box was placed in the mower's path, after the mower had begun its mowing operation.

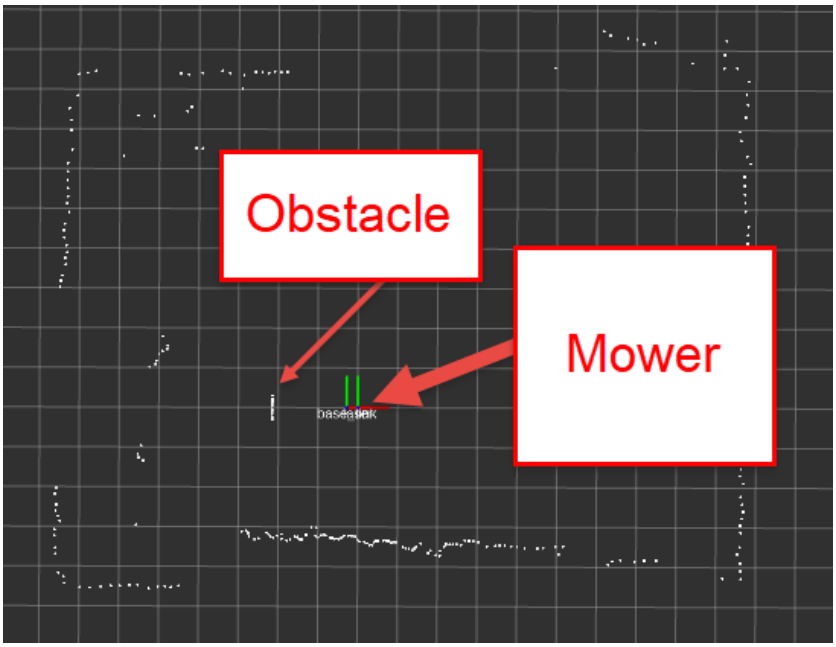


Figure 140. LIDAR scan of obstacle detection test with 24" square box

Figure 141 shows another instance of the obstacle detection test. This time a 12" cardboard tube was placed in the mower's path. In both instances, the mower successfully detected the obstacle and stopped mowing operations approximately 60" ahead of the obstacle.

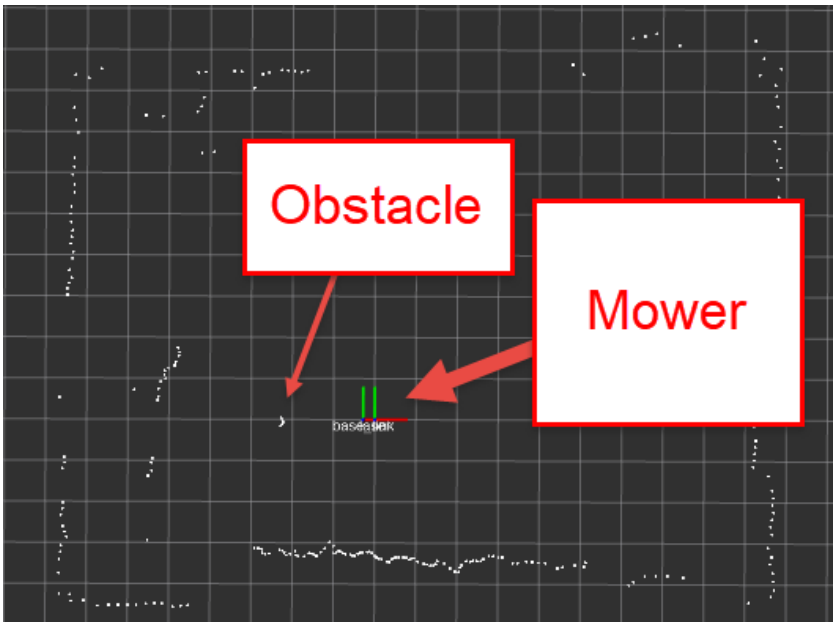


Figure 141. LIDAR scan of obstacle detection test with 12" round tube

7.3.3 Stop and wait

The stop and wait functionality is used in conjunction with the dynamic obstacle detection routine. As shown in Figure 139, if a dynamic obstacle is detected, the current navigational logic for both long and short-term paths is cancelled. The goal is cancelled in the ROS system to avoid the ROS navigational system from attempting to avoid the obstacle. Specifically cancelling the job in ROS prevents any further navigation from occurring within the ROS environment. Once this occurs, the coverage grid node, which is the node currently executing the navigation path, enters its own loop to wait for the object to clear. It checks the current LIDAR unit readings once-per-second to see if the object is no longer present. If it is present, the system continues to wait indefinitely. If the object clears the path, the stop and wait routine calls back into the executePath function. The last executing PathCommand index is tracked in the _stopWaitIndex variable and used to resume a waypoint for the previously executing goal. Since the ROS

navigation system is capable of navigation from the current position to the destination waypoint, no further instructions are required.

Chapter 8

Summary and Conclusions

The goal of this project was to conduct the research and development necessary to build a cost-effective robotic system capable of autonomously mowing a complex yard in an efficient manner. Creating the various hardware and software systems required for this project was much more time-consuming than originally anticipated. There were many issues encountered over the course of the project that slowed progress, chief among them being uncooperative weather conditions. Learning about the machining process in order to create the aluminum mechanical components on the mill and lathe also took substantial time. Dealing with electrical noise generated by the engine and the electromagnetic noise generated by the mower deck and their effects on the computer system also proved challenging. These issues were difficult to track down as they did not always manifest themselves as consistent failures in the system. Often the failures did not stop a system from functioning, but rather caused the signals used by a system to become erratic, unstable, or noisy.

In the end, these various difficulties were overcome resulting in a system that was able to *mostly* achieve my original goal. I say mostly because the original intent was to autonomously mow a *complex* yard. While I was able to get the system to the point of autonomously mowing a simple rectangular area, I would not characterize the area as being complex. It did have inclines and stone walls as boundaries, but it did not have any significant obstacles present inside the work area. However, I was able to successfully test the dynamic obstacle detection routines in the live environment. Additionally, I was able to successfully test the navigational planning routines in a more complex virtualized

environment. Testing these planning routines in a more complex actual environment on the mower itself will be left to future work, as discussed below in section 8.2.

There were three main aspects to this project. The first involved creating custom computer hardware while the second required the creation of software components to interface to these new hardware components. The third aspect involved the creation of software to intelligently control the overall system in order to produce the desired results. To this end, I developed custom computer hardware and software systems and then integrated them into a commercial-grade lawnmower. These custom systems included a laser distancing system, a wheel encoder system, and navigational system. Additionally, I had to mechanize the mower to allow computer control of various mower hardware components. This included the engine, the drive system, the parking brake, and the mower deck.

8.1 Contributions

There were several hardware and software systems developed for use in this project that represent the unique contributions of this thesis. The mechanical and electronic aspects of these systems were built using readily available off-the-shelf components. Since the hardware was interfaced with the ROS environment through the loose coupling offered by custom ROS nodes, the hardware discussed in this thesis can be reused in other projects. Additionally, the software components were also created with reuse in mind and therefore have their various configuration parameters publicly exposed. These parameters allow the hardware and software components to be tailored for a variety of use-case scenarios beyond the one defined in this thesis. The key contributions of this thesis are detailed in the following subsections.

8.1.1 Throttle control system

A throttle control system was developed for the ROS environment that allows the control and management of the Kawasaki gasoline engine used in the Bad Boy Outlaw XP mower. The ROS node created for this purpose was designed to be easily adapted to other gasoline-powered engines. The PID algorithm control parameters were exposed as configurable input values to the ROS node. Additionally, the Hall Switch sensor inputs and servo PWM outputs are also configurable. These configuration points allow the throttle control node to be customized for use with other engines using differing tachometer inputs. This allows for the reuse of the throttle control node in a variety of other projects.

8.1.2 LIDAR system

The custom laser-based distancing system I developed for this project represents a low-cost alternative to much more expensive units used in most academic projects. Three LIDAR units were built and combined into one virtual unit through a custom ROS node created for this project. This virtual device allowed for a 360-degree view around the perimeter of the mower. The trade-off between my system and more expensive systems is primarily one of speed. However, as demonstrated in this project, even though the custom laser system operated at a slower rate, it proved sufficient given the slow operating speed of the mower. Additionally, the use of stepper motors allowed precise control of the laser. This precise control allowed interleaving of the forward-facing lasers, which provided faster detection of objects than could be done through the virtual laser itself.

8.1.3 Odometry system

A novel use of diametrically aligned magnets allowed for the creation of a robust wheel encoder system. These wheel encoders output quadrature signals, which allowed the custom software to track both the speed and direction of wheel rotation while being impervious to dirt and dust. The hardware concepts created for this system can be easily adapted to other hardware platforms. Additionally, the ROS node created to interface with the hardware can easily be configured for use in other environments. This thesis also demonstrated that fusing the wheel encoder data with the X-orientation of the IMU resulted in more accurate wheel odometry, especially during turning operations. The overall wheel odometry ROS node is configurable and componentized to enable reuse in other projects requiring accurate odometry for robots using track-based or zero-turn drive systems.

8.1.4 Path planning system

The path planning system implemented for this project followed the Boustrophedon algorithm. While the Boustrophedon algorithm has been implemented for many other projects, this thesis implemented it using a unique virtualized representation of the actual environment. This virtualized environment is configurable in size, the number and location of obstacles as well as the mower size. The ability to transfer the planned path out of the virtualized environment and back into the actual environment is also reusable by other robot implementers using ROS. Additionally, the virtualized environment is not limited to the Boustrophedon algorithm and allows for any number of path planning algorithms to be virtually tested and then transferred to the live environment.

8.2 Future Work

The sensor fusion taking place between the wheel encoders and the IMU currently only makes use of the IMU's X-orientation. This fusion offers the biggest improvements during turning operations, as this is when wheel slip is most prevalent. However, the IMU also outputs non-fused data in the form of acceleration. I believe fusing the linear acceleration data from the IMU with the wheel encoders will help detect and minimize slip errors during normal, non-turning forward travel. The slip detected during forward travel was minimal, but I believe the error would increase as the mower encountered more challenging work areas with greater grades. Making use of additional IMU data could help minimize such errors.

While the use of the IMU greatly increased the accuracy of the odometry system, especially during turning operations, it involved the manual calibration of the turning parameters. For instance, in order to make a 90-degree turn, the mower system could not simply wait until the IMU reported a 90-degree change in direction. It had to stop ahead of the 90-degrees of change in order to allow for the mechanical systems to come to rest at 90-degrees. How much ahead of the actual 90-degree mark stopping needed to commence depended greatly upon environmental conditions. The coefficient of friction values outlined in table 10 were used to help approximate this value. As a future enhancement, it would be beneficial to create an auto-calibration process to determine this value. A process could be developed to utilize the ground type (grass or gravel) along with the speed of the mower and the discrete accelerometer and gyroscope sensor values from the IMU to accurately compute the stop ahead angular value.

The process of making a work area map is largely manual at this time. It requires the user to manually manipulate the navigational map image file, annotating it with a red bounding box to define the desired work area, then saving the changes as a secondary image file. The navigational map is constructed using the SLAM algorithm. As discussed, SLAM is a process for both localizing and mapping an area. It should be possible to use data generated during the SLAM process and create the work-area bounding box while the navigational map is generated. However, as discussed in chapter 6, in order to get a high-quality map, I had to drive the mower around more than just the perimeter of the work area. This additional driving would make it more challenging to determine the work area boundary automatically.

One possible solution would be to keep the navigational map creation process as-is and instead introduce a second stage to this process for defining the work area. For example, the first stage could remain as driving around both the perimeter of the work area and the middle of the work area in order to create a high-quality navigational map. The user could then signal the completion of this stage by pressing the autonomous mode button on the control panel. Doing so could then put the system into the second stage work area boundary creation process. Then the user could drive the perimeter of the desired work area while the system uses the AMCL techniques outlined in chapter 7 to localize the mower and record the work area boundary automatically. While still a two-step process, it would greatly simplify the creation of the work area map and remove the manual process of file manipulation.

Another area of future improvements involves the trimming process. The path planner uses rectangles to bound obstacles that should be avoided during mowing

operations. Not all objects are rectangular in shape, for example, trees and rocks. Using rectangles to bound obstacles simplifies the path planning and also makes for cleaner overall cuts. However, it would be nice to have the mower return over the mowed area and perform one or more clean-up passes around these obstacles to remove any further grass that might exist outside the buffer zone. Additionally, these clean-up passes could be performed at a slower speed thereby allowing the buffer zone to be reduced from the current 14" mark. The buffer zone itself could be defined in relation to the speed and current function of the mower (e.g. Trim vs. Mow).

There is room for optimization in the path planning process when moving from one work cell to another. Currently, the planner moves from one work cell to another simply using the order in which the edges between cells were detected. When the first work cell has been completely covered, the system moves the mower to the second work cell detected, even if another work cell adjoining the currently completed work cell might be present. A planner that used a Reeb graph to represent the topology of multiple work cells could be used to reorganize the travel plan based on some set of weighted cost criteria. For instance, minimizing distance or perhaps the opportunity to clean-up and trim around nearby areas during travel could be used as the cost criteria. The virtual planner could then be used to run simulations on potential paths to find paths that minimize cost.

Appendix A.

Glossary

AMCL

Adaptive Monte Carlo Localization.

Encoder

A device that converts one type of information to another. As used in this project, an encoder converts angular position to digital pulses using magnets or optics.

FSM

Finite State Machine.

Hz

Hertz is a unit of frequency represents the number of times per second some event occurs.

I2C

The Inter-Integrated Circuit Protocol designed by Phillips in the early 1980's to allow communication between Master/Slave devices.

IMU

Inertial Measurement Unit, provides a way of measuring linear and angular motion in a self-contained unit. Linear motion is measured using accelerometers while angular motion is measured using gyroscopes. Magnetometers are often present as well to provide heading information.

LIDAR

Light Detection and Ranging is a method of measuring distance by emitting pulses of light and measuring the amount of time it takes for these pulses to return to their source.

NMEA

National Marine Electronics Association.

Odometry

A method of using sensor data to estimate position over time.

PPM

Portable Pix Map.

PID

Proportional-Integral-Derivative controller. A PID controller is a system that implements a control loop based on a feedback to maintain a given set point value.

PGM

Portable Grey Map.

PTO

Power-Take-Off, provides power from the engine to the mower deck.

PWM

Pulse Width Modulation, in the current context it is a method of varying the amount of power delivered to a motor while still maintaining constant voltage. This is done by varying the duty cycle of the binary digital signal. A 50% duty cycle is a square wave where 50% of the time the signal is high and 50% of the time the signal is low. The speed of a motor can be increased by increasing the amount of time the signal is high. The speed of the motor can be decreased by decreasing the amount of time the signal is high.

ROS

Robot Operating System, an open-source Linux-based meta-operating system for controlling robots and handling sophisticated programming tasks such as path planning. ROS is service-based, meaning each module can be coded separately allowing for easy extension and algorithm implementation sharing.

RPM

Revolutions Per Minute.

SiP

System-in-a-Package.

SPST

Single-Pole-Single-Throw.

SLAM

Simultaneous Localization and Mapping.

VDC

Volts Direct Current.

References

AAeon. (2018). UP board 4 GB RAM + 64 GB eMMC. Retrieved February 12, 2018 from https://up-shop.org/up-boards/44-up-board-4gb-ram-64-gb-emmc.html

Acar, E.U., Choset, H., Rizzi, A.A., Atkar, P.N., Hull, D. (2002). Morse decompositions for coverage tasks. *The International Journal of Robotics Research*. 21(4):331-344.

Acroname. (n.d.). Hokuyo UTM-30LX-EW Scanning Laser Rangefinder. Retrieved June 3, 2016, from http://acroname.com/products/HOKUYO-UTM-30LX-EW?sku=R354-UTM-30LX-EW

Adafruit. (2018a). Adafruit Metro Mini 328 – 5V 16MHz. Retrieved February 12, 2018 from https://www.adafruit.com/product/2590

Adafruit. (2018b). MCP9808 High Accuracy I2C Temperature Sensor Breakout Board. Retrieved February 12, 2018 from https://www.adafruit.com/product/1782

Adafruit. (2018c). Adafruit Ultimate GPS Breakout. Retrieved February 12, 2018 from https://www.adafruit.com/product/746

Adafruit. (2018d). Adafruit 9-DOC Absolute Orientation IMU Fusion Breakout BNO055. Retrieved February 12, 2018 from https://www.adafruit.com/product/2472

Adafruit. (2018e). Adafruit TB6612 1.2A DC/Stepper Motor Driver Breakout Board. Retrieved February 12, 2018 from https://www.adafruit.com/product/2448

Adafruit. (2018f). Miniature Slip Ring – 12mm diameter, 6 wires, max 240V @ 2A. Retrieved February 18, 2018 from https://www.adafruit.com/product/775

Åkesson, B. M., Jørgensen, J. B., Poulsen, N. K., & Jørgensen, S. B. (2007). A tool for kalman filter tuning. *Computer Aided Chemical Engineering 17th European Symposium on Computer Aided Process Engineering,* 859-864. doi:10.1016/s1570-7946(07)80166-0

Alamo Industrial. (n.d.). Traxx RF. Retrieved December 17, 2017 from http://www.alamo-industrial.com/TRAXX/Index.asp

Amazon. (2018a). JBtek 4 Channel DC 5V Relay Module. Retrieved February 12, 2018 from https://www.amazon.com/JBtek-Channel-Module-Arduino-Raspberry/dp/B00KTEN3TM

Amazon. (2018b). NEMA 14 Motor Step Motor 1.8deg Bipolar 12V 0.4A 14N/cm/20oz.in 35x35x26mm. Retrieved February 18, 2018 from https://www.amazon.com/gp/product/B00PNEPQ8E/ref=oh_aui_search_detailpage?ie=UTF8&psc=1

Arduino wiring source. (December 2015). Retrieved December 29, 2017, from https://github.com/arduino/Arduino/blob/master/hardware/arduino/avr/cores/arduino/wiring.c

Bad Boy Mowers. (2015). Outlaw XP Zero Turn Mower Owner's, Service & Parts Manual. Batesville, AK: Bad Boy Mowers.

Beauregard, Brett. (April 2011). Improving the Beginner's PID. Retrieved July 7, 2017, from http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/

Bosch. (November 2014). BNO055 Intelligent 9-axis absolute orientation sensor data sheet. Bosch Document # BST-BNO055-DS000-12.

Bosch. (n.d.). Indego 400 Connect - Robotic lawnmowers. Retrieved December 17, 2017 from http://www.bosch-garden.com/gb/en/garden-tools/garden-tools/indego-400-connect-3165140828284-209530.jsp

Bourke, Paul. (n.d.). PPM / PGM / PBM image files. Retrieved September 16, 2017, from http://paulbourke.net/dataformats/ppm/

Cenek, P., Jamieson, N., McLarin, M. (May 2016). Frictional Characteristics of Roadside Grass Types. Retrieved June 12, 2017, from https://saferroadsconference.com/wp-content/uploads/2016/05/Peter-Cenek-Frictional-Characteristics-Roadside-Grass-Types.pdf

Choset, H., Lynch, K. M., & Hutchinson, S. (2005). *Principles of robot motion: theory, algoritms, and implementations*. Cambridge, MA: Bradford.

Choset, H., & Pignon, P. (1998). Coverage Path Planning: The Boustrophedon Cellular Decomposition. *Field and Service Robotics,* 203-209. doi:10.1007/978-1-4471-1273-0_32

Creer, Terry. (2007). DIY Radio Controlled Lawnmower. Retrieved December 17, 2017, from http://members.iinet.net.au/~tnpshow/RCLM/intro.htm.

Cub Cadet. (n.d.). RG3. Retrieved December 17, 2017 from http://www.cubcadetturf.com/rg3

Daltorio, K. A., Rolin, A. D., Beno, J. A., Hughes, B. E., Schepelmann, A., Branicky, M. S., Quinn, R., Green, J. M. (2010). An obstacle-edging reflex for an autonomous lawnmower. *IEEE/ION Position, Location and Navigation Symposium*. doi:10.1109/plans.2010.5507339

Dave Z JR. (May 2014). New Remote control lawnmower exmark walk behind. Retrieved December 17, 2017, from http://www.youtube.com/watch?v=NU9SWmq42jY

Dellaert, F., Fox, D., Burgard, W., & Thrun, S. (n.d.). Monte Carlo localization for mobile robots. *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. doi:10.1109/robot.1999.772544

Dempsey, Paul. (2007). *How to repair Briggs & Stratton Engines*. New York, NH: The McGraw-Hill Companies.

Deublin. (2018). The Basics of Electrical Slip Rings. Retrieved February 18, 2018 from http://www.deublin.com/assets/1/7/NEW_Deublin_TheBasicsElectricalSlipRings_HR_11.37.51_AM.pdf

Dimension Engineering. (April 2012). Sabertooth 2x12 User's Guide. Retrieved February 12, 2018 from https://www.dimensionengineering.com/datasheets/Sabertooth2x12.pdf

Evans, Lee. (2011, September 19). *Remote Control Mowbot Explained* [Video file]. Retrieved June 16, 2017, from http://www.youtube.com/watch?v=Q7crpT4FELQ

EvaTech. (n.d.). Hybrid Goat Robot 32. Retrieved December 17, 2017 from http://evatech.net/PRODUCT.php?ID=674

Ferguson, Michael. (n.d.). rosserial: ROS Serial Package Summary. Retrieved December 23, 2017 from http://wiki.ros.org/rosserial

Galceran, E., & Carreras, M. (2013). A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61(12), 1258-1276. doi:10.1016/j.robot.2013.09.004

García-Alegre, M., Ribeiro, A., Garcia-Perez, L., Martinez, R., Guinea, D. (2001). Autonomous Robot in Agriculture Tasks. *Proceedings 3rd European Conference on Precision Agriculture ECPA.*

Garmin. (2018a). LIDAR-Lite v3. Retrieved February 12, 2018 from https://buy.garmin.com/en-US/US/p/557294

Garmin. (2018b). Lidar Lite v3 Operation Manual and Technical Specifications. Retrieved February 12, 2018 from http://static.garmin.com/pumac/LIDAR_Lite_v3_Operation_Manual_and_Technical_Specifications.pdf

Gonzalez, Daniel J. (2012, November 14). Quadrature Encoders in Arduino, done right. Retrieved June 16, 2017 from http://yameb.blogspot.com/2012/11/quadrature-encoders-in-arduino-done.html

Grayson, Paul. (2016, January 21). Do robotic mowers dream of electric turf? Retrieved June 3, 2016 from http://www.golfcourseindustry.com/article/do-robotic-mowers-dream-of-electric-turf/

Grisetti, G., Stachniss, C., & Burgard, W. (2007). Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics*, 23(1), 34-46. doi:10.1109/tro.2006.889486

Guofang, G., & Zhengzhong, L. (2012). Design of Heating Furnace Temperature Control System Based on Fuzzy-PID Controller. *Lecture Notes in Electrical Engineering Information Engineering and Applications,* 1463-1469. doi:10.1007/978-1-4471-2386-6_196

Hall, R. (2016, March 7). The Rise of Robotic Lawn Mowers. Retrieved April 25, 2016, from http://www.turfmagazine.com/technology/robotic-lawn-mowers/

Held, T. (2015, May 11). Briggs & Stratton exploring robotic lawnmower. Retrieved April 25, 2016 from http://www.bizjournals.com/milwaukee/news/2015/05/11/briggs-stratton-exploring-robotic-lawnmower.html

Hirschmuller, H. (2005). Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR05)*. doi:10.1109/cvpr.2005.56

Husqvarna. (n.d.). Husqvarna Robotic Lawnmowers. Retrieved December 17, 2017 from http://www.husqvarna.com/us/products/robotic-lawn-mowers/automower-450x/967622605/

Irus Mowers. (n.d.). Deltrak 2.5. Retrieved December 17, 2017, from http://irus-mowers.co.uk/deltrak-2-5/

Jarvis, Ray. (2001, November). A Tele-Autonomous Heavy Duty Robotic Lawn Mower. *Australian Conference on Robotics and Automation.* 157-161.

johndavid400. (2010, March 24). Arduino R/C Lawnmower. Retrieved December 17, 2017 from http://www.instructables.com/id/Arduino-RC-Lawnmower/

John Deere. (n.d.). Tango E5 Series - Robotic Mower. Retrieved December 17, 2017 from http://www.deere.com/en_INT/products/equipment/robotic_mower/tango_e5_series_II/tango_e5_series_II.page

K&J Magnetics. (2018). Magnetization Direction for Neodymium Magnets. Retrieved February 12, 2018 from https://www.kjmagnetics.com/magdir.asp

Kent323is. (2015, August 28). *ATTC RC throttle test* [Video file]. Retrieved on December 4, 2017 from

https://www.youtube.com/watch?v=26hGIFCDxZI&list=PLWuiU1ejGipmLMME0u4HEjSNpCyGB1dO8&index=13

Kreinar, E. (2013). *Filter-Based Slip Detection for a Complete-Coverage Robot* (Electronic Thesis or Dissertation). Retrieved from https://etd.ohiolink.edu/

Latombe, J. C. (1993). Robot Motion Planning. Boston: Kluwer Academic.

LawnBotts. (n.d.). LawnBott LB300EL Robotic Lawn Mower. Retrieved December 17, 2017 from http://www.lawnbotts.com/lawnbott/LawnBott_LB300EL_Robotic_Lawn_Mower.html

LeddarTech. (2016, January 19). *Leddar Sensors Road Test for ADAS and Autonomous Driving* [Video file]. Retrieved April 26, 2016, from https://www.youtube.com/watch?v=5rtpwDmFguo

Leonard, J., & Durrant-Whyte, H. (1991). Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation, 7*(3), 376-382. doi:10.1109/70.88147

Levy, Simon D. (n.d.). The Extended Kalman Filter: An Interactive Tutorial for Non-Experts. Retrieved July 10, 2017 from http://home.wlu.edu/~levys/kalman_tutorial/

Low Power Lab. (2018). All about Moteino: Introduction. Retrieved February 12, 2018 from https://lowpowerlab.com/guide/moteino/

Melexis. (2018). Unipolar Hall-effect switch – Low sensitivity. Retrieved February 12, 2018 from https://www.melexis.com/en/product/US5881/Unipolar-Hall-Effect-Switch-Low-Sensitivity

Mertz, C., Navarro-Serment, L. E., Maclachlan, R., Rybski, P., Steinfeld, A., Suppé, A., . . . Gowdy, J. (2012). Moving object detection with laser scanners. *Journal of Field Robotics,30*(1), 17-43. doi:10.1002/rob.21430

Michael, M., Salmen, J., Stallkamp, J., & Schlipsing, M. (2013). Real-time stereo vision: Optimizing Semi-Global Matching. *2013 IEEE Intelligent Vehicles Symposium (IV)*. doi:10.1109/ivs.2013.6629629

Motion Control Products. (n.d.). Stepper Motor Theory. Retrieved March 21, 2017 from http://www.motioncontrolproducts.com/pages/applications-stepper-motor.php

Mr2Tuff2. (2013, May 28). *Remote Controlled Lawn Mower* [Video file]. Retrieved from https://www.youtube.com/watch?v=6oBZGzPNQoE

National Robotics Engineering Center (NREC). (n.d.). Golf Course Mowing project. Retrieved April 25 2016, from http://www.nrec.ri.cmu.edu/projects/toro/

Oksanen, T., Visala, A. (2007, July). Path Planning Algorithms for Agricultural Machines. *Agricultural Engineering International: the CIGR Ejournal*. Manuscript ATOE 07 009. Vol. IX.

OpenCV. (2017). Open Source Computer Vision Library. Retrieved February 12, 2018 from https://opencv.org/about.html

Paques, Alexis. (2016, December 6). LIDARLite v3 ENHANCED. Retrieved March 22, 2017 from https://github.com/AlexisTM/LIDAREnhanced

Phidgets. (n.d.). Encoder Primer. Retrieved January 15, 2018 from https://www.phidgets.com/docs/Encoder_Primer

PJRC. (2018). Teensy USB Development Board. Retrieved February 12, 2018 from https://www.pjrc.com/store/teensy32.html

Pradhan, R., Majhi, S. K., Pradhan, J. K., & Pati, B. B. (2017). Performance Evaluation of PID Controller for an Automobile Cruise Control System using Ant Lion Optimizer. *Engineering Journal, 21*(5), 347-361. doi:10.4186/ej.2017.21.5.347

Quigley, M., Gerkey, B., & Smart, W. D. (2015). Programming Robots with ROS. Sebastopol, CA: O'Reilly Media.

Raymond, Eric, S. (2016, January). NMEA Revealed. Retrieved February 12, 2018 from http://www.catb.org/gpsd/NMEA.html

Reitman, J. (2015, September 4). Robotic mowers paid off for Pennsylvania club. Retrieved April 25, 2016 from http://www.turfnet.com/news.html/_/robotic-mowers-paid-off-for-pennsylvania-club-r565

Remote Mowers. (n.d.). TRX-44-PRO Commercial Slope Mower. Retrieved December 17, 2017 from http://www.remotemowers.com/trx-44-pro

Robomow. (n.d.). RS630 - Robomow Robotic Lawnmower. Retrieved December 17, 2017 from http://usa.robomow.com/shop/mowers/large-gardens-en-us/rs-630-5/

ROS. (2018a, February 14). ROS sensor_msgs/LaserScan Message. Retrieved February 26, 2018 from http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html

ROS. (2018b, February 14). ROS sensor_msgs/Imu Message. Retrieved February 26, 2018 from http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html

Santos, J. M., Portugal, D., & Rocha, R. P. (2013). An evaluation of 2D SLAM techniques available in Robot Operating System. *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. doi:10.1109/ssrr.2013.6719348

Schaenzle, Jordan. (2016, June 28). PID – Helping Computers Behave More Like Humans. Retrieved January 1, 2018 from https://spin.atomicobject.com/2016/06/28/intro-pid-control/

Servo City. (2018). HS-5646WP Servo. Retrieved February 12, 2018 from https://www.servocity.com/hs-5646wp-servo

Silicon Labs. (2017). CP2014 Single-Chip USB-to-UART Bridge. Retrieved February 25, 2018 from https://www.silabs.com/documents/public/data-sheets/cp2104.pdf

Slamtec. (2018). RPLIDAR A1 Laser Range Scanner. Retrieved February 12, 2018 from http://www.slamtec.com/en/lidar/a1

Solvelight Robotics. (n.d.). Denna L1000 Robot Lawn Mower. Retrieved December 17, 2017, from http://www.solvelight.com/product/denna-l1000-robot-lawn-mower

Stewart, D. (2016, February 29). Using autonomous & robotic mowers to solve the labor crisis. Retrieved April 25, 2016 from http://landscapemanagement.net/the-future-workforce-using-autonomous-robotic-mowers-to-solve-the-labor-crisis/

SuperDroid Robots. (n.d.). 2WD 66in Lawn Mower – WC. Retrieved December 17, 2017, from www.superdroidrobots.com/shop/item.aspx/2wd-66in-lawn-mower-wc/1985/

Thrun, S., Burgard, W., & Fox, D. (2010). Probabilistic Robotics. Cambridge, MA: MIT Press.

usb.org. (n.d.). Cyclical Redundancy Checks in USB. Retrieved April 4, 2017 from http://www.usb.org/developers/docs/whitepapers/crcdes.pdf

Warren, J. D. (2010, March 11). Arduino R/C Lawnmower. Retrieved April 24, 2016 from http://www.instructables.com/id/Arduino-RC-Lawnmower/

Warren, J. D. (2012, December 18). Build a Lawnbot 400. Retrieved April 24, 2016 from http://makezine.com/projects/lawnbot400/

Wollerton, M. (2015, August 17). Robot lawn mowers might be on iRobot's horizon. Retrieved April 25, 2016 from http://www.cnet.com/news/robot-lawn-mowers-might-be-on-irobots-horizon/

Woodall, William. (2012, May 25). *ROSCon 2012 – "Moe" The Autonomous Lawnmower* [Video file]. Retrieved April 26, 2016 from https://www.youtube.com/watch?v=eQecc04lDbc

Worx. (n.d.). Landroid M. Retrieved December 17, 2017 from http://www.worx.com/landroid-robotic-lawn-mower-wg794.html

Wu, X., Xu, M., & Wang, L. (2013). Differential speed steering control for four-wheel

independent driving electric vehicle. *2013 IEEE International Symposium on Industrial Electronics*. doi:10.1109/isie.2013.6563667